I μ-controllori ARM: la logica RISC

I μ-controllori ARM (ARM è l'acronimo di Advanced RISC Machines Ltd) sono molto interessanti perché basati su di una architettura di funzionamento diversa da quella vista fino ad ora.

Gli ARM appartengono a quella classe di μ-controllori detti RISC, Reduced Instruction Set computer (l'8051 e gli altri processori visti sono detti CISC, Complex Instr. Set Comp).

E' spesso difficile cercare di capire la filosofia di funzionamento di un μ-controllore. Spesso non è neanche fondamentale conoscerla. Basta sapere sfruttare quelle che sono le caratteristiche del dispositivo.

Tuttavia la conoscenza dell'architettura RISC è utile per potere essere in grado di selezionare e/o discriminare il migliore candidato al progetto a cui si sta lavorando.

Va subito fatta l'osservazione che RISC, malgrado la definizione, non significa tanto avere a che fare con meno istruzioni, ma bensì con una architettura circuitale che sia molto semplice nel consentire di ottenere maggiore compattezza, velocità di operazione ed esecuzione delle istruzioni in un singolo colpo di clock. La lunghezza delle istruzioni viene mantenuta costante in modo da garantire una prevedibilità nei tempi di esecuzione delle istruzioni.

La semplice filosofia che sta dietro l'architettura RISC si può riassumere nella figura sotto.

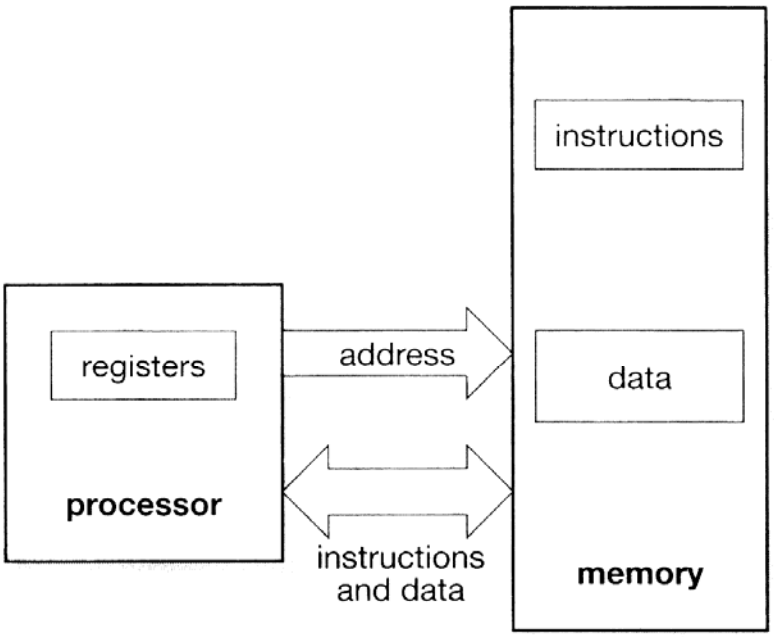


Figure 1.1 The state in a stored-program digital computer.

$FF..FF_{16}$

Abbiamo una unità centrale, il processore, che è la parte intelligente del sistema. Un insieme di registri consente di movimentare i dati all'interno della CPU.

I dati e le istruzioni vengono caricati dalla memoria.

Non si possono quindi eseguire operazioni dirette sui data in memoria, ma solo tra i registri interni.

$00..00_{16}$

La struttura RISC

Per essere semplice e veloce l'architettura RISC prevede:

- ✓ Un solo formato di istruzione da 32 bit. In questo modo nella singola istruzione si può prevedere l'indicazione di più operandi, o comandi (le istruzioni dei CISC variano in numero di byte);
- ✓ Filosofia load-store, ovverosia le istruzioni che operano sui dati lo fanno solo attraverso i registri interni al cuore del processore. In seguito i dati possono essere trasposti nella memoria (nei CISC un operando può essere un dato della memoria);
- ✓ Un banco di 32 registri a 32-bit ognuno utilizzabile per un qualsiasi scopo e hard-wired, cioè indipendente dagli altri, così che la struttura load-store possa agire in modo efficiente (nei CISC i registri hanno spesso scopi definiti).

L'organizzazione RISC prevede:

- ✓ La decodifica delle istruzioni è hard-wired (in confronto con la struttura a micro-code di molti CISC);
- ✓ L'esecuzione delle istruzioni è organizzata secondo la filosofia pipeline (la struttura CISC spesso non permette questo approccio perché le istruzioni hanno numero di byte variabile);
- ✓ Esecuzione in un solo colpo di clock (nei CISC abbiamo visto che spesso servono più colpi di clock).

Svantaggio della struttura RISC:

- ✓ Una minore densità di codice rispetto alla struttura CISC. in una struttura RISC c'è ridondanza in molte istruzioni.

La pipeline nei RISC 1

Nei RISC ogni istruzione è organizzata in modo che le azioni vengano svolte seguendo sempre una medesima successione, eventualmente lasciando un buco, o NOP, se quel particolare passo non serve.

Questo fa si che si possa evitare di aspettare che una istruzione sia completamente eseguita, prima di iniziare la successiva. Mentre una istruzione è in corso di esecuzione se ne potrebbe iniziare un'altra. Questo approccio si dice organizzazione a pipeline .

Nell'ARM7 classicamente si adotta una pipeline a 3 livelli:

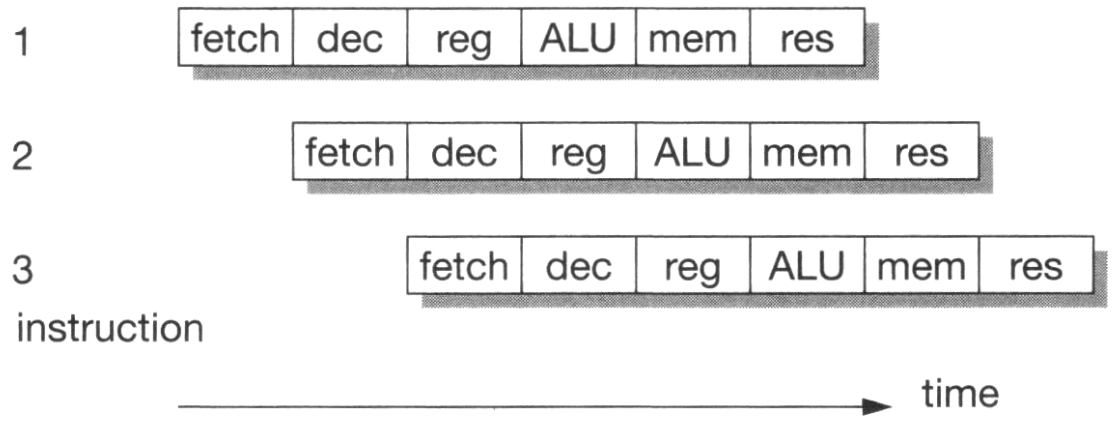


Figure 1.13 Pipelined instruction execution.

L'organizzazione a pipeline deve essere organizzata all'atto della realizzazione hardware del dispositivo.

L'organizzazione a 3 livelli dell'ARM evita molte delle conflittualità possibili. Nelle ultime generazioni di ARM si è arrivati anche a 5 livelli di pipeline.

Nelle operazioni di scrittura/lettura dalla memoria si ottiene un buon beneficio dalla struttura a pipeline.

In termini di velocità si hanno buoni benefici.

Occorre però fare attenzione che l'organizzazione a pipeline può portare a colli di bottiglia, se non accuratamente progettata. Altrimenti è compito dello sviluppatore dell'applicazione fare in modo che nell'esecuzione del programma si evitino situazioni di stallo.

La pipeline nei RISC 2

Usare troppi livelli di pipeline può portare problemi. Uno di questi è il read-before-write, ovverosia una istruzione di un livello alto nella pipeline ha bisogno di un dato che l'istruzione a livello inferiore non ha ancora prodotto, ciò che porta al così detto stallo.

Oppure ci può essere il problema di un salto, condizionato o meno, che potrebbe portare ad un'inefficienza.

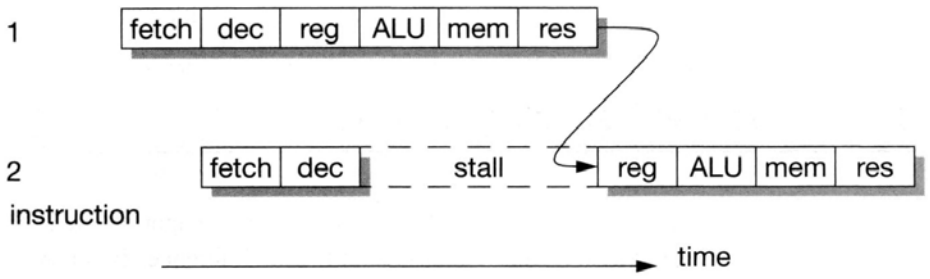


Figure 1.14 Read-after-write pipeline hazard.

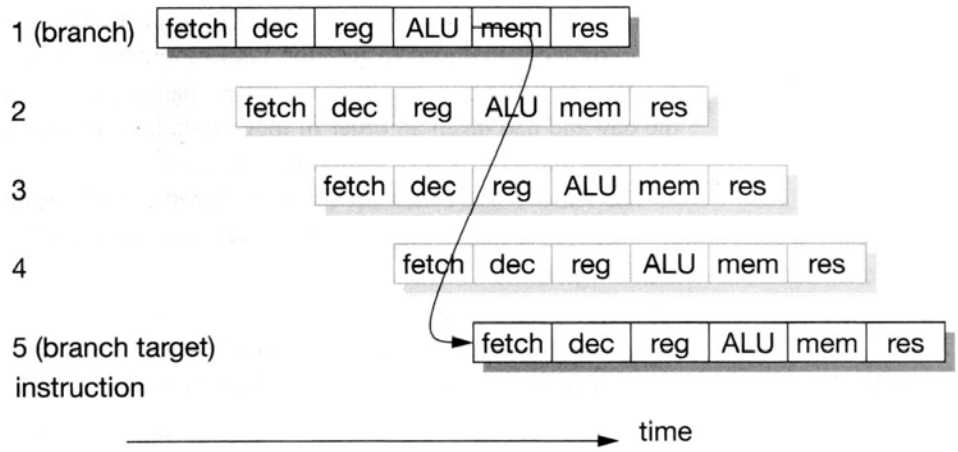
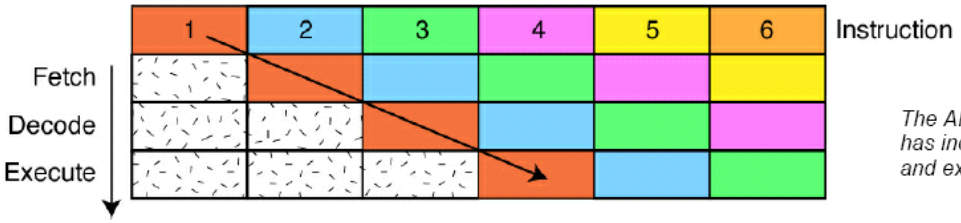


Figure 1.15 Pipelined branch behaviour.

Questi problemi portano ad una livellazione del livello di pipeline. 3 livelli di pipeline, come nell'ARM7, sono agevolmente implementabili. 5 livelli di pipeline, come nell'ARM9, rappresentano certamente un bel progresso.

La pipeline nei RISC 3

L'ARM esegue istruzioni in un solo colpo di clock. Secondo questa concezione la pipeline dovrebbe essere ridondante. In realtà il collo di bottiglia si ha quando le istruzioni ed i dati vanno presi dalla memoria e caricati nella CPU. In questa situazione la pipeline a 3 stadi consente di potere mantenere il singolo colpo di clock sia che si debba eseguire un'istruzione nella CPU che di trasferimento di dati e/o istruzioni.



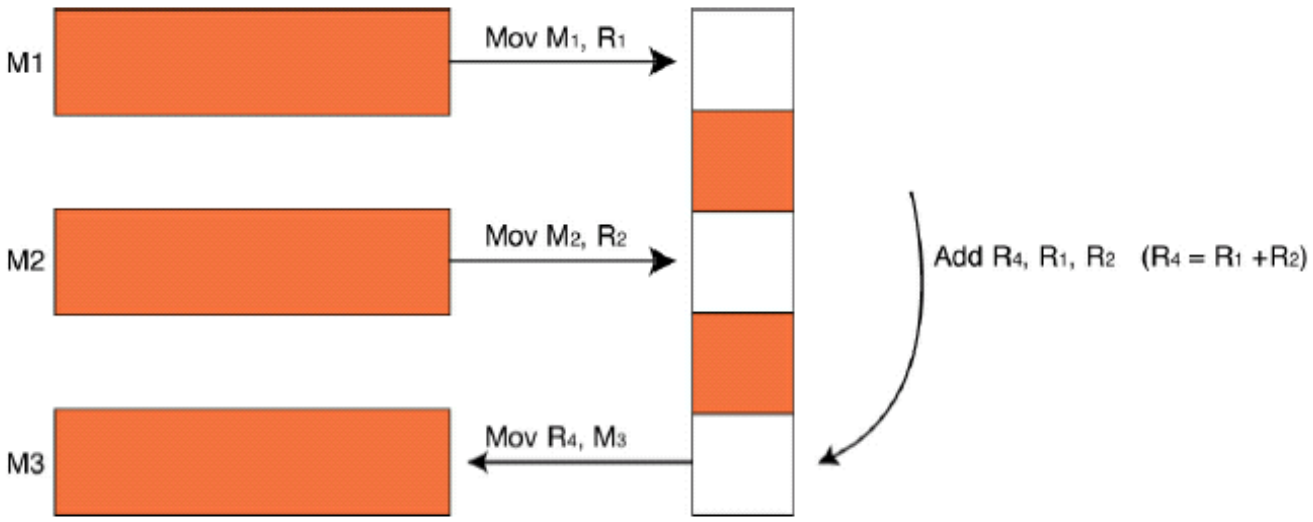
The ARM7 three-stage pipeline has independent fetch, decode and execute stages

La struttura a pipeline deve essere un parametro progettuale che deve essere tenuto in conto all'atto della realizzazione della struttura, non è solo una proprietà firmware.

Importante: siccome il PC punta a 2 istruzioni successive a quella in svolgimento occorre fare attenzione al suo attuale valore quando si vogliono eseguire indirizzamenti relativi al PC.

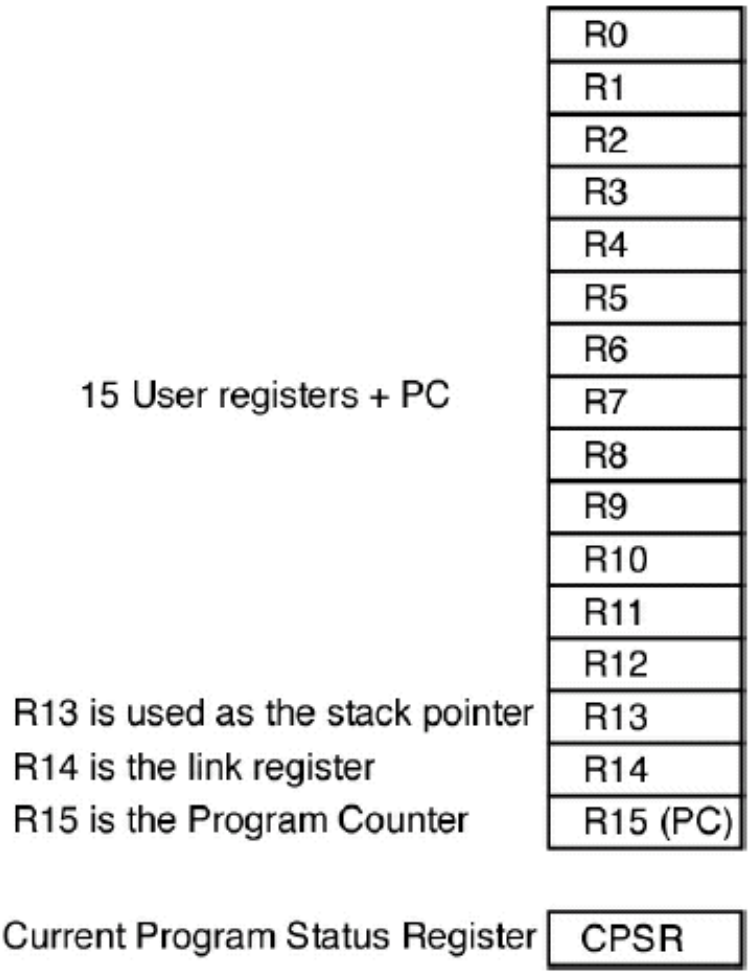
Si è detto che l'ARM ha una struttura load-and-store. Vale a dire che i dati dalla memoria, per essere elaborati, vanno prima caricati nel set di registri che compongono la CPU.

Esempio: M3= M1 + M2



I registri nella CPU dell'ARM nella modalità standard, o dell'utilizzatore

Veniamo ora al set di registri. Non è niente di complicato in un ARM: si tratta, nella modalità di operazione standard, di 16 registri ed un registro di stato:



I registri da **R0** a **R12** sono di utilizzo completamente generale: possono essere usati per caricare dati da/verso la memoria ed utilizzati per operazioni matematiche.

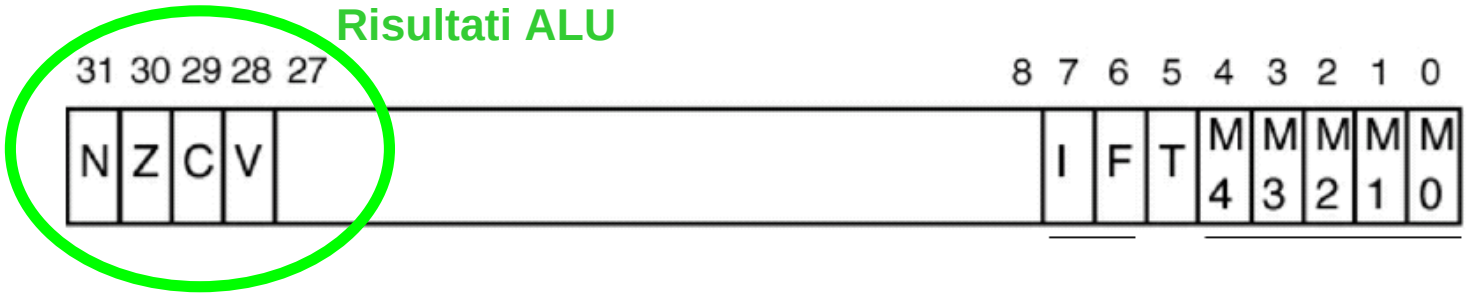
I registri da **R13** a **R15** svolgono invece uffici particolari, come indicato.

Infine abbiamo il **Current Program** (vedremo perché Current) **Status Register**.

Il **CPSR** contiene gli essenziali **flags** riguardanti la descrizione del risultato dell'ultima operazione eseguita nell'ALU.

I primi 8 bit hanno anche un significato differente.

Si nota come la ridondanza viene impiegata qui: su 32 bits se ne usano solo 12.



La struttura ARM 1

Nell'ARM la memoria dati e programmi non è necessariamente esterna al chip. E' solamente ritenuta esterna alla CPU. Il banco di registri è dove ruota lo svolgimento del percorso dei dati.

Innanzi tutto si può vedere come il **PC** sia auto-incrementato e sia connesso sia al registro indirizzi che al banco dei registri. Infatti il **PC** è il registro **R15**.

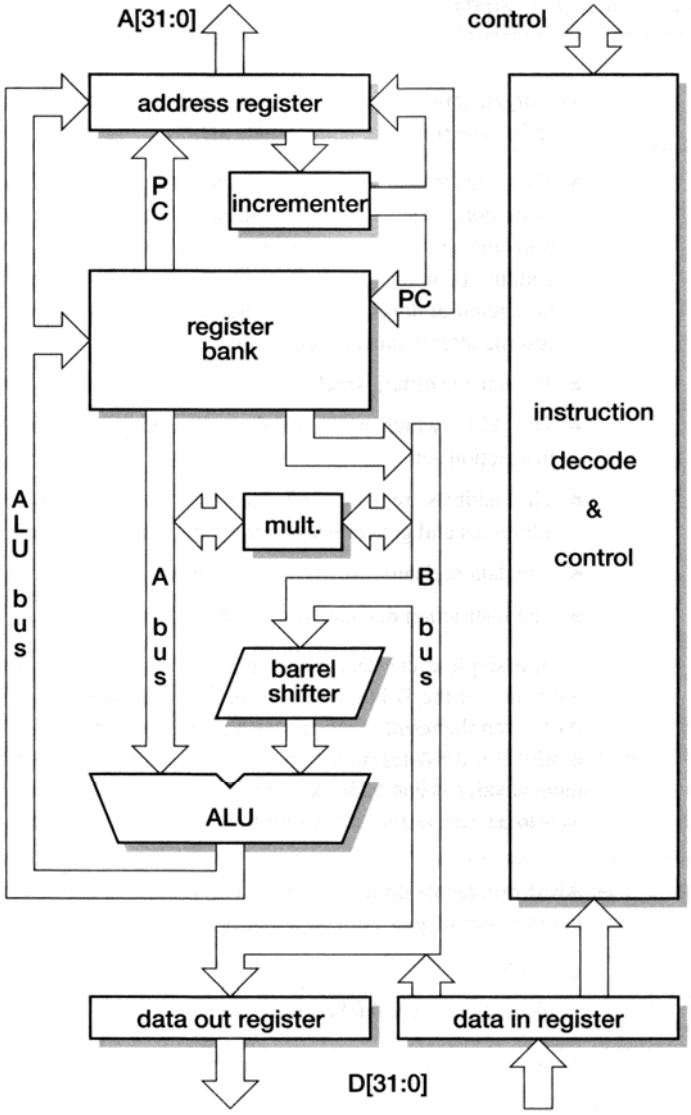


Figure 4.1 3-stage pipeline ARM organization.

L'eventuale indirizzo del nuovo dato deve uscire dalla ALU.

L'ARM si avvicina al comportamento di un DSP.

Per cui possiede una unità di **moltiplicazione** HW.

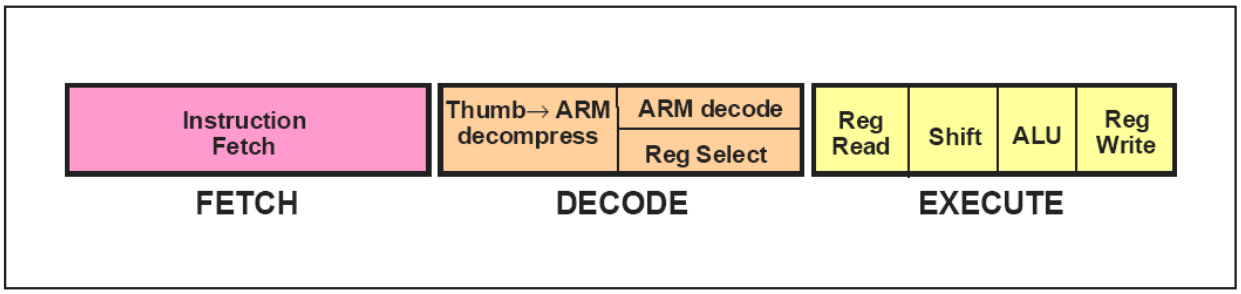
E pure un **barrel shifter**, che consente di ruotare le parole di memoria in un singolo colpo di clock (si ricorda che ogni slittamento a sinistra corrisponde ad una moltiplicazione per 2, uno a destra una divisione per 2).

Si noti un aspetto fondamentale: l'ARM in origine, ARM7, possiede una struttura Harvard modificata in quanto non c'è una separazione tra il bus dati ed istruzioni all'interno della CPU.

La struttura ARM 2

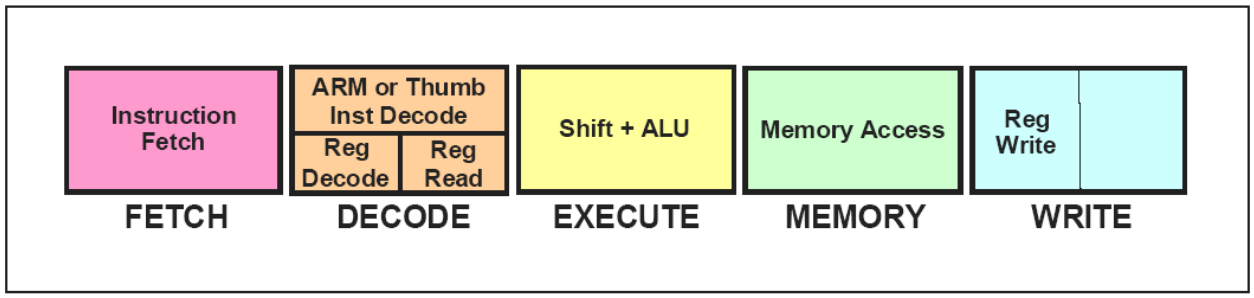
La pipeline a 3 stadi e l'avere in comune il bus per i dati e le istruzioni è una prerogativa delle versioni più datate degli ARM, dette ARM7.

Figure 1 : The ARM7TDMI core and ARM7TDMI-S core pipeline



Con l'ARM9 il livello di pipeline è stato portato a 5 stadi ed è stata introdotta anche la separazione del bus dati da quello delle istruzioni. La velocità della macchina è quindi ancora più estesa.

Figure 2 : The ARM9TDMI core pipeline



Esecuzione di un'istruzione nell'ARM: matematica tra 2 registri 1

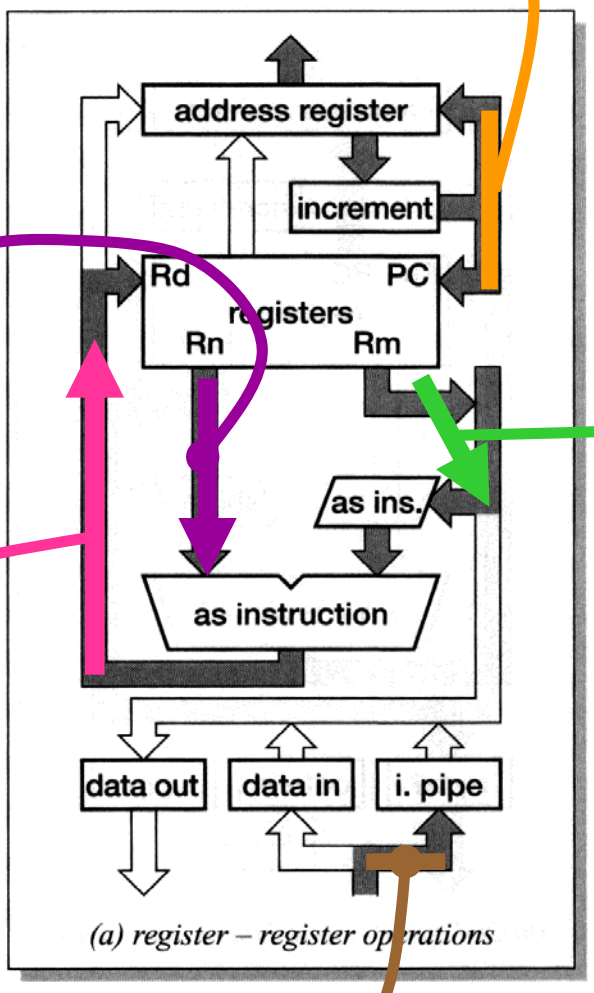
Il processamento di 2 numeri avviene seguendo 2 modalità: operazione tra 2 registri o tra un registro ed una valore diretto ad 8 bit contenuto nell'istruzione.

Siccome l'istruzione è eseguita in un singolo colpo di clock, il secondo operando non può essere una cella di memoria.

Si deve osservare che mentre si svolge l'operazione nella ALU il PC viene già aggiornato.

In una istruzione di processamento un operando è sempre un registro (Rn)

Il secondo operando può essere un registro (Rm), che può essere soggetto ad un'operazione di shift.



Il risultato dell'operazione ritorna al banco di registri.

La nuova istruzione entra dalla pipeline (i. pipe)

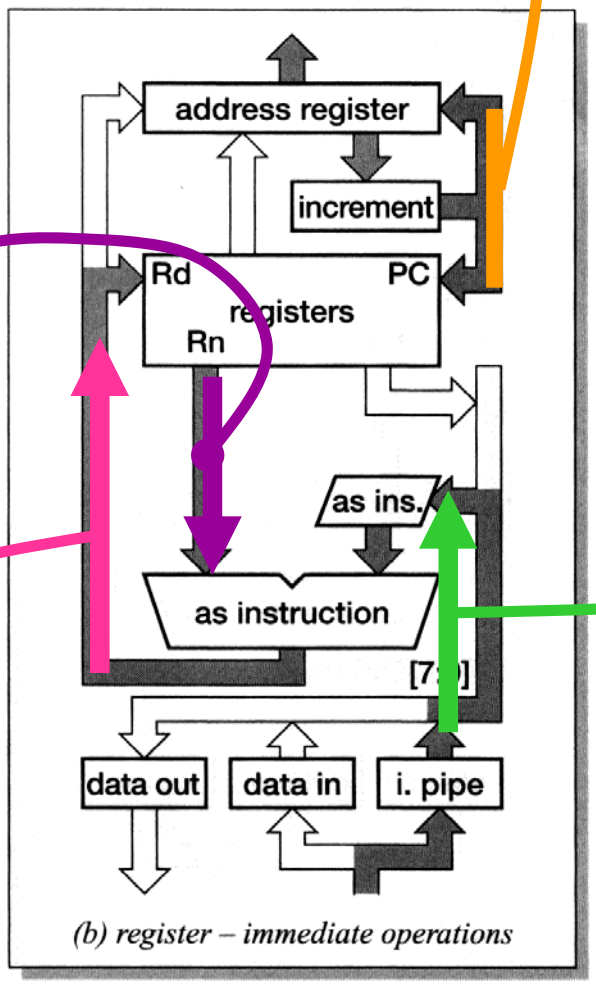
Esecuzione di un'istruzione nell'ARM: matematica tra 2 registri 2

Si deve osservare che mentre si svolge l'operazione nella ALU il PC viene già aggiornato.

In una istruzione di processamento un operando è sempre un registro (Rn)

Il secondo operando può essere un numero immediato proveniente dalla istruzione, che può essere soggetto ad un'operazione di shift.

Il risultato dell'operazione ritorna al banco di registri.



(b) register – immediate operations

Una tale organizzazione consente di eseguire istruzioni a 3 registri. Per esempio:

```
ADD     r0, r1, r2          ; r0 := r1 + r2
ADC     r0, r1, r2          ; r0 := r1 + r2 + C
SUB     r0, r1, r2          ; r0 := r1 - r2
SBC     r0, r1, r2          ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2          ; r0 := r2 - r1
RSC     r0, r1, r2          ; r0 := r2 - r1 + C - 1
```

Esecuzione di un'istruzione nell'ARM: trasferimento dati 1

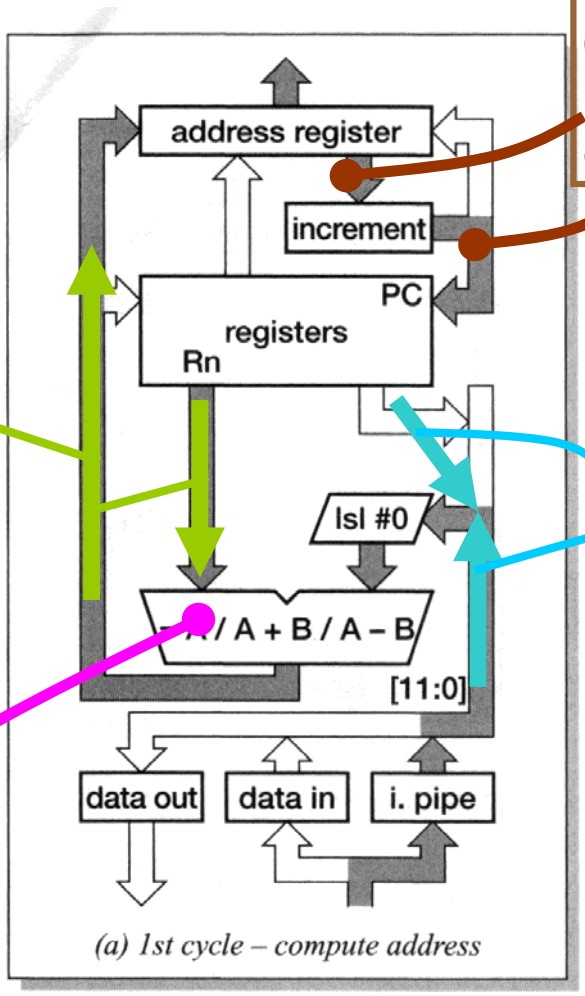
Il trasferimento dei dati da/verso la memoria avviene in 2 steps. Nel primo si forma l'indirizzo, nel secondo si legge/scrive il dato.

I° ciclo: l'indirizzo

Il contenuto del registro della predente istruzione, il PC, viene incrementato e posto nel PC. In effetti il registro indirizzi fa anch'esso da pipeline.

L'indirizzo del dato viene posto nel registro indirizzi. Questo indirizzo proviene da un registro, dopo essere passato dall'ALU, dove può essere manipolato.

All'interno dell'ALU il contenuto del registro può essere modificato sommando o sottraendo un offset che proviene da un altro registro, o da 12 bit del campo istruzioni. L'operazione di shift non può essere applicata.



(a) 1st cycle – compute address

Per guadagnare tempo, nelle istruzioni in cui è necessario incrementare l'indirizzo, già in questa fase è eseguita questa funzione.

Esecuzione di un'istruzione nell'ARM: trasferimento dati 2

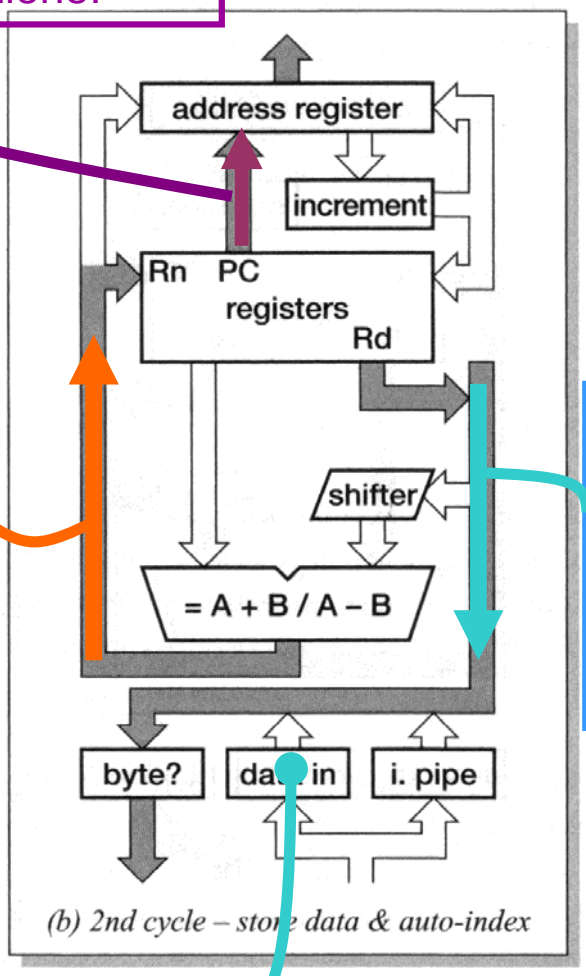
L'operazione di trasferimento del dato avviene in 2 cicli. Nel secondo ciclo il dato viene letto/scritto all'indirizzo posto sul bus nel primo ciclo.

II° ciclo: il trasferimento del dato

Il PC viene inviato nel registro indirizzi per la ricerca, fetch, della prossima istruzione.

L'indirizzo valutato precedentemente viene messo nel registro, per futuri usi.

Il dato viene a questo punto posto in uscita. Il dato può anche essere inviato come byte, se necessario.



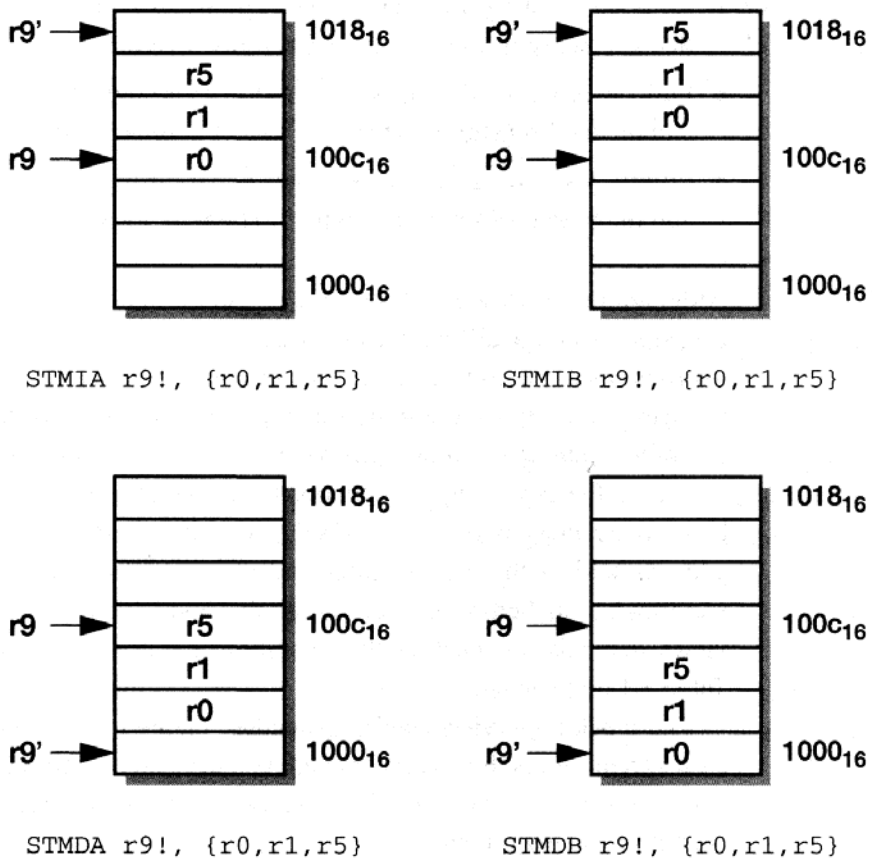
Se l'operazione fosse stata di load il dato proveniente dalla memoria sarebbe stato posto nel data in register. A questo punto un ulteriore ciclo sarebbe stato necessario per trasferire il dato nel registro finale.

Esecuzione di un'istruzione nell'ARM: trasferimento dati 3

Negli ARM, addirittura, si riescono ad eseguire istruzioni indicizzate, in cui si hanno trasferimenti multipli di registri:

```
STMFD   r13!, {r2-r9}      ; save regs onto stack
LDMIA   r0!,  {r2-r9}
STMIA   r1,   {r2-r9}
LDMFD   r13!, {r2-r9}      ; restore from stack
```

L'effetto è sostanzialmente quello di avere una memorizzazione a stack:

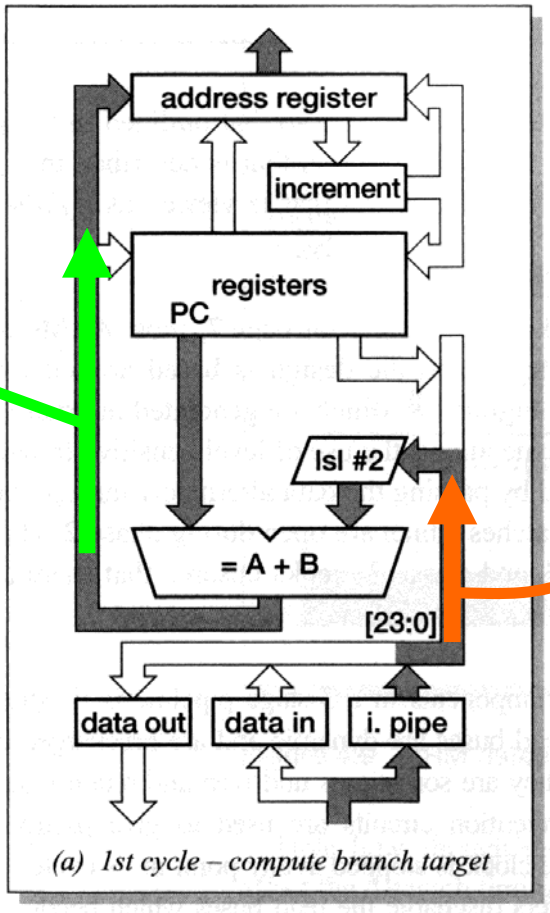


Istruzione di salto 1

L'istruzione di salto è svolta in 3 passi.

I° passo: il nuovo indirizzo è creato

Al contenuto del PC viene sommato un offset proveniente dai primi 24 bit del campo istruzione.



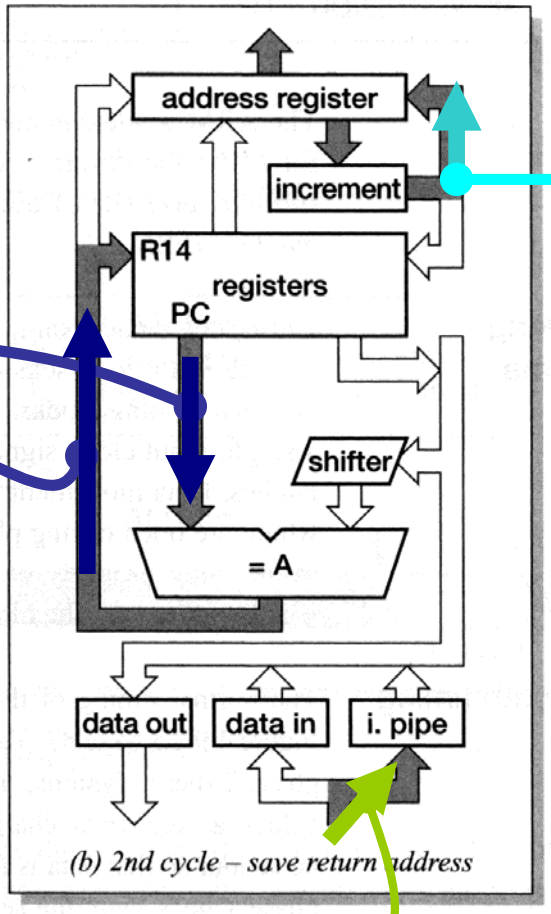
I primi 24 bit dell'istruzione vengono shiftati di 2 posizioni a sinistra per avere allineamento.

Istruzione di salto 2

Se il salto è ad un sottoprogramma l'indirizzo di ritorno è posto nel link register nella seconda fase dell'istruzione.

Il terzo ciclo sarà la lettura vera e propria dell'istruzione.

II° passo: aggiornamento del link register (se richiesto)



Incremento del PC.

Il registro di ritorno, se è chiamato un sottoprogramma, viene aggiornato ora nel link register (R14)

Fetch della istruzione successiva: l'effetto della organizzazione della pipeline.

Istruzione di salto 3

Nel dettaglio l'organizzazione binaria dell'istruzione di salto condizionato/incondizionato si ha con 4 bit di condizione, 3 bit che indicano il tipo di condizione, 24 bit di che rappresentano il punto a cui saltare rispetto al valore attuale del PC.

Infine abbiamo il bit L: il PC viene messo in R14 per il ritorno.

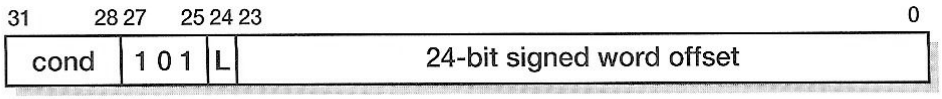


Figure 5.3 Branch and Branch with Link binary encoding.

Ecco qualche esempio di salto condizionato/incondizionato, come viene rappresentato nel linguaggio assembly:

An unconditional jump:

```
        B       LABEL   ; unconditional jump ..
        ..
LABEL   ..              ; .. to here
```

To execute a loop ten times:

```
        MOV     r0, #10 ; initialize loop counter
LOOP    ..
        SUBS    r0, #1  ; decrement counter setting CCs
        BNE     LOOP    ; if counter <> 0 repeat loop..
        ..              ; .. else drop through
```

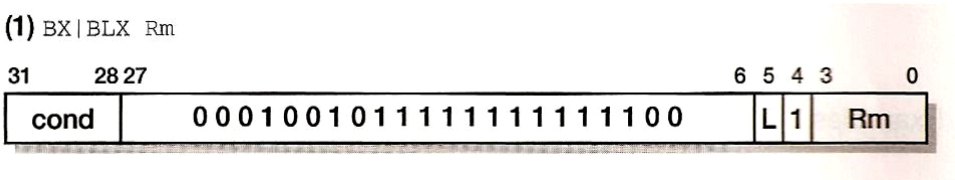
To call a subroutine:

```
        ..
        BL      SUB     ; branch and link to subroutine SUB
        ..              ; return to here
        ..
SUB     ..              ; subroutine entry point
        MOV     PC, r14 ; return
```

Istruzione di salto 4

Una ulteriore modalità di salto è quella in cui l'indirizzo a cui il PC deve andare è contenuto in un registro.

Il salto può ancora essere condizionato. In questo caso il bit L ci dice se occorre salvare il PC in R14 per garantire il ritorno da un sottoprogramma.



Infine possiamo anche avere un salto incondizionato in cui si salta alla modalità Thumb. L'indirizzo del salto sono 24 bit di offset da somare/sottrarre al PC:

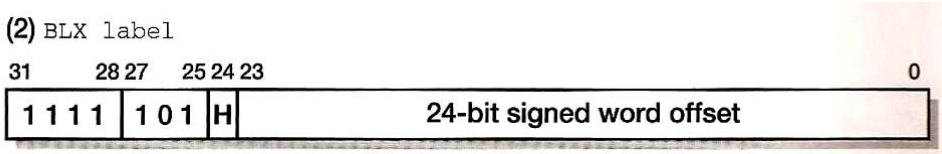


Figure 5.4 Branch (with optional link) and exchange instruction binary encodings.

Esempio:

```
1:      B{L}X{<cond>} Rm
2:      BLX <target address>
```

'<target address>' is normally a label in the assembler code; the assembler will generate the offset (which will be the difference between the word address of the target and the address of the branch instruction plus 8) and set the H bit if appropriate.

An unconditional jump:

```
        BX      r0      ; branch to address in r0,
                        ; enter Thumb state if r0[0] = 1
```

A call to a Thumb subroutine:

```
        CODE32          ; ARM code follows
        ..
        BLX     TSUB    ; call Thumb subroutine
```

Tempistica

E' interessante notare che nell'ARM i registri non operano sul fronte di salita o discesa del clock, ma sul livello, per esempio alto. Per questa ragione dal clock principale viene creato una seconda fase che non si sovrappone alla prima.

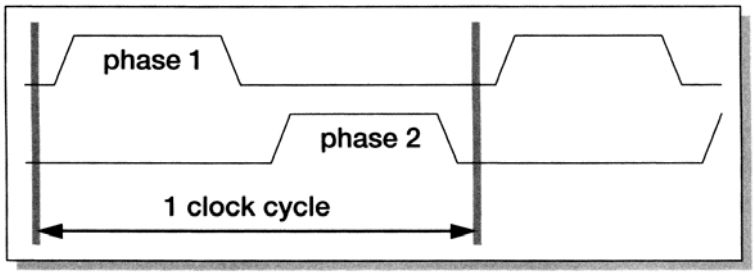


Figure 4.8 2-phase non-overlapping clock scheme.

Ci saranno quindi registri che diventeranno trasparenti durante la prima fase del clock e registri che saranno trasparenti durante la seconda fase del clock, senza sovrapposizione.

Va notato che la vita interna dell'ARM è cadenzata dalla presenza di registri, trasparenti all'utilizzatore, che consentono il trasferimento dei dati seguendo le tempistiche opportune.

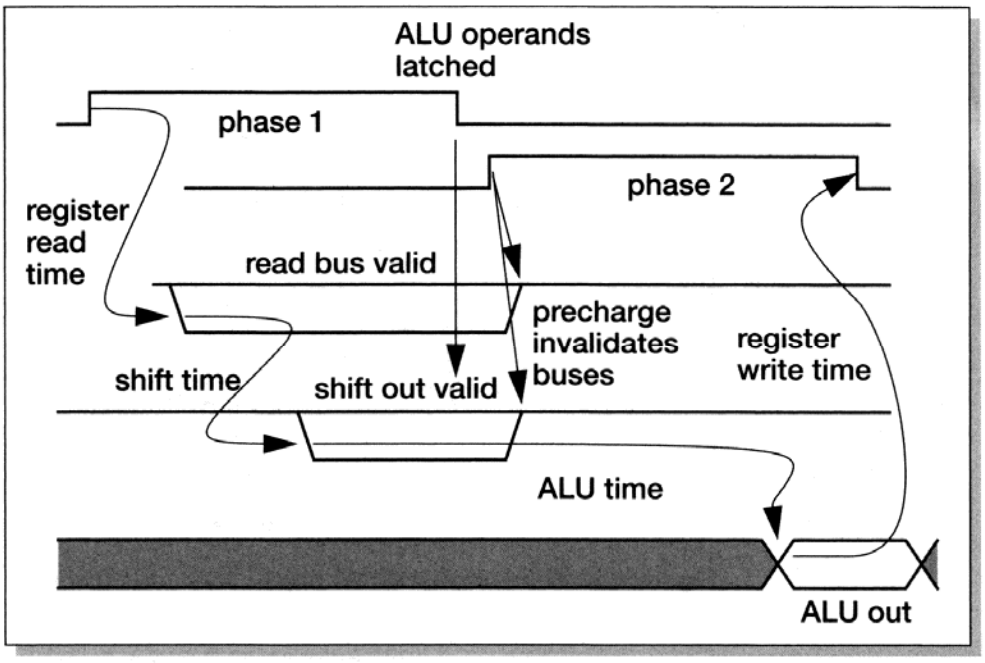


Figure 4.9 ARM datapath timing (3-stage pipeline).

Modalità di funzionamento: le eccezioni (exceptions) 1

L'ARM ha 6 modalità di funzionamento (**exceptions**) con le quali si fronteggiano tutte le possibili situazioni, compresi gli interrupt. Le exceptions sono sostanzialmente dei sottoprogrammi che vengono chiamati sulla base di determinate condizioni che accadono nei vari stati del sistema.

Ad ogni modalità è associato un set di registri aventi parti in comune e distinte rispetto alla modalità usuale (Standard o User) di funzionamento.

In questo modo la gestione di questi sottoprogrammi è svolta in modo più celere.

Vi sono 6 diversi stati operativi. I registri comuni a tutti gli stati sono quelli non colorati. I registri colorati sono invece visibili solo allo stato pertinente.

Il concetto è molto semplice: si vuole fare in modo che nella chiamata del sottoprogramma si minimizzino i passi da svolgere per salvare le variabili nello stack, per consentire il loro successivo ripristino. Abbiamo una funzione in più rispetto ad una chiamata standard.

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7_fiq	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Modalità di funzionamento: le eccezioni (exceptions) 2

Allo stesso modo che nelle interruzioni, ad ogni eccezione è assegnato un vettore, un indirizzo del codice programmi, da dove si lancerà il sottoprogramma vero e proprio.

Exception	Mode	Address
Reset	Supervisor	0x00000000
Undefined instruction	Undefined	0x00000004
Software interrupt (SWI)	Supervisor	0x00000008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C
Data Abort (data access memory abort)	Abort	0x00000010
IRQ (interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Abbiamo una priorità nella gestione di queste modalità operative:

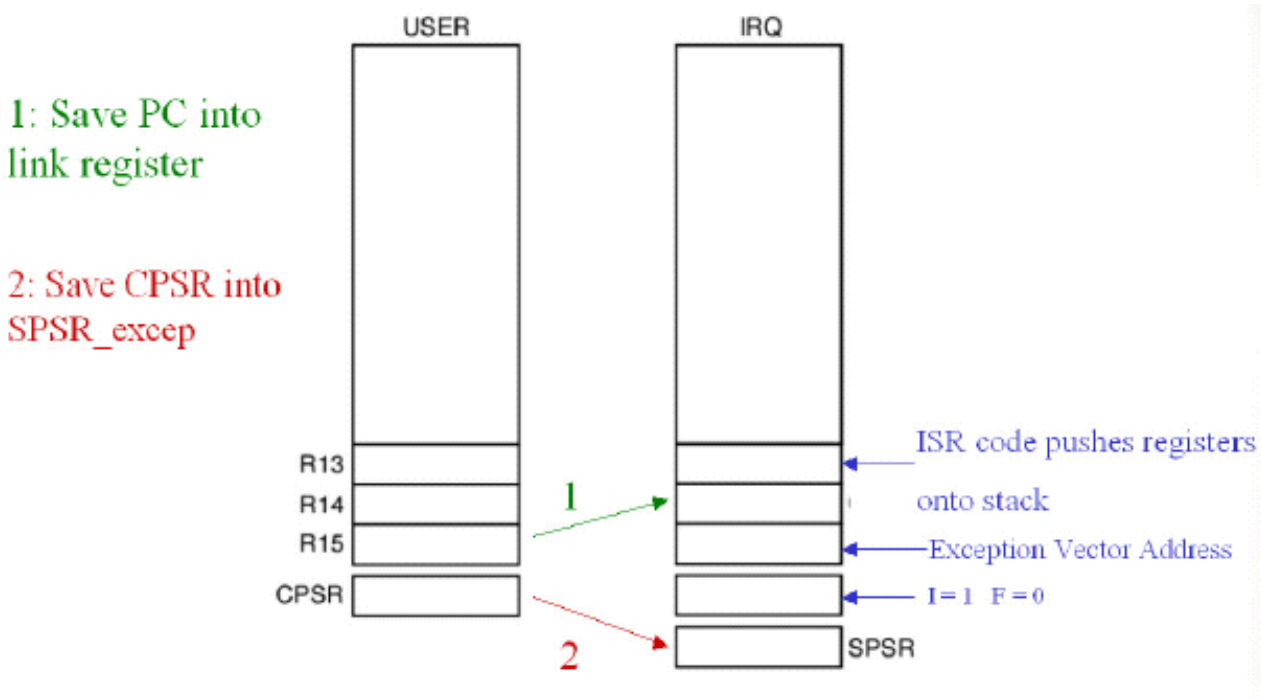
Priority	Exception
Highest 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Lowest 6	Undefined instruction SWI

Modalità di funzionamento: le eccezioni (exceptions) 3

Cosa succede in definitiva?

Il gioco viene svolto essenzialmente a livello dei 3 registri R14, R15 ed SPSR.

- Quando la modalità deve essere inserita il **PC** (**R15**) viene salvato automaticamente e contemporaneamente in **R14** (**LR**, l'indirizzo di ritorno) della modalità finale.
- Contemporaneamente, nel **PC** viene inserito il valore del vettore contenuto in **R15** della modalità finale.
- Il **CPSR** viene salvato, contemporaneamente, in un registro chiamato **SPSR_modalità**. Allo stesso tempo la modalità di interruzione viene annullata.

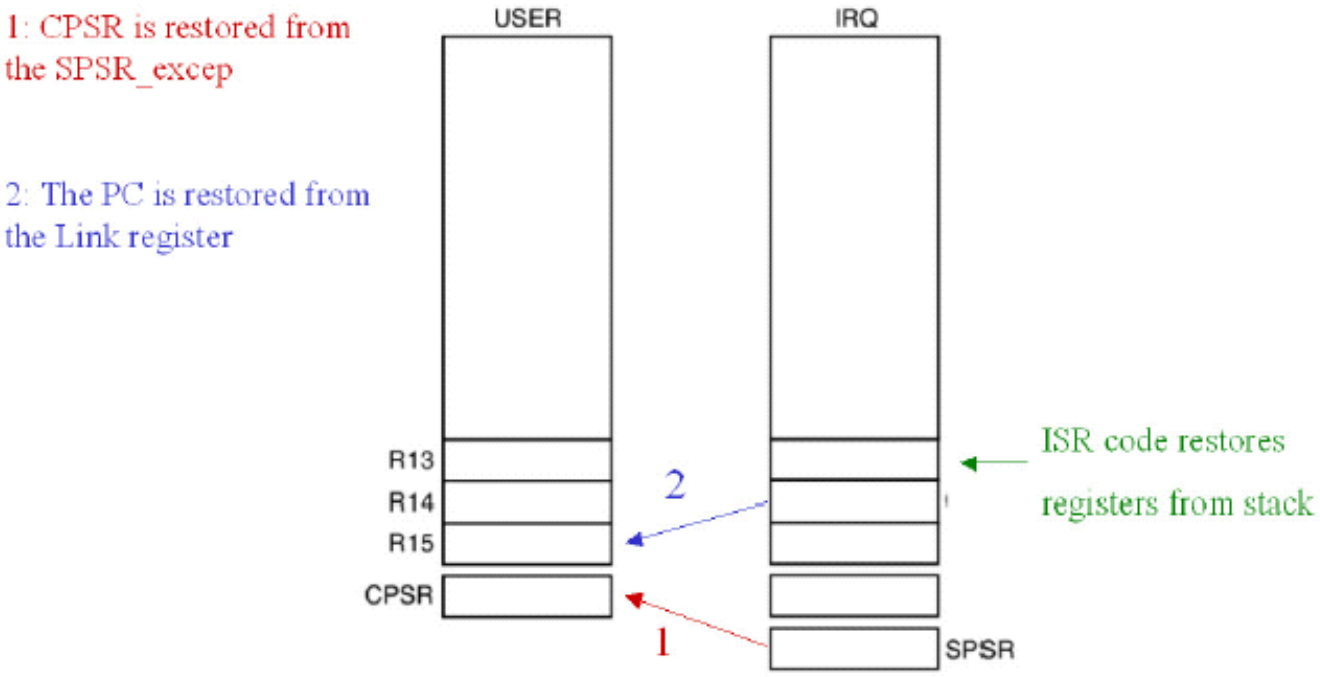


Il registro R13 della modalità inserita è lo stack pointer che può essere sfruttato per conservare i registri in comune con la modalità di partenza, se necessario.

Modalità di funzionamento: le eccezioni (exceptions) 4

Il ritorno dalla modalità inserita al funzionamento precedente segue la procedura inversa:

- Il **PC** viene aggiornato con il **LR** della modalità inserita
- Il **CPSR** viene aggiornato con il contenuto del registro **SPSR_modalità**.

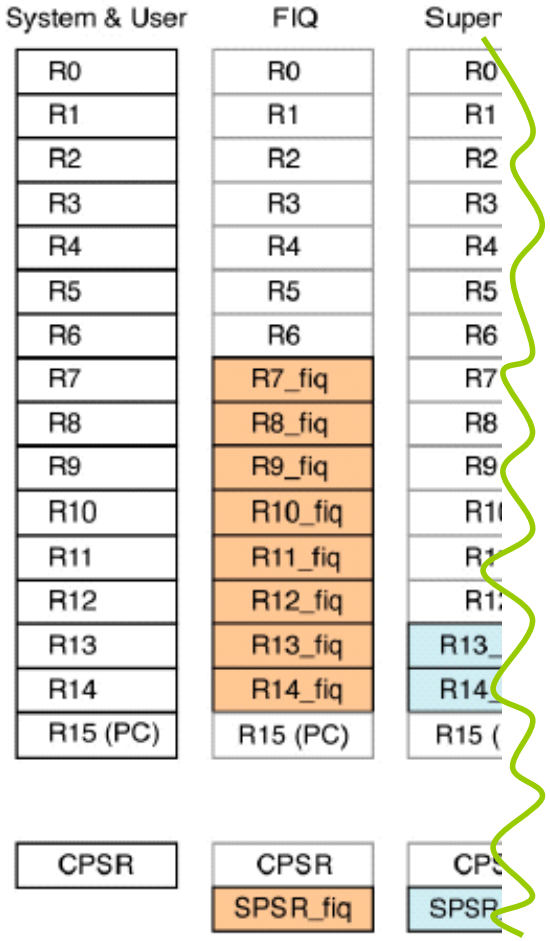


Ovviamente se dei registri sono stati preservati durante la chiamata devono essere ripristinati.

Tra tutte le modalità hanno indipendenti i registri **R13** (stack pointer), **R14** (Link Register) e **l'SPSR**, ed in comune tutti gli altri eccetto la modalità **FIQ**, Fast Interrupt Request.

Modalità di funzionamento: le eccezioni (exceptions) 5

Nella modalità FIQ anche i registri da R7 ad R12 sono comuni solo alla FIQ stessa.



In tale modo si cerca di risparmiare tempo nel dovere impilare nello stack registri: si fornisce l'opportunità di avere a disposizione un certo numero di registri indipendenti.

Il concetto che sta alla base è che in genere è difficile che debbano verificarsi più di una interruzione per volta. Il sistema è ottimizzato per tale circostanza. Tuttavia, l'opportunità di gestione di più interruzioni in cascata è comunque implementabile. Ovviamente, in tale caso, l'efficienza si ridurrebbe a quella di un microcontrollore standard.

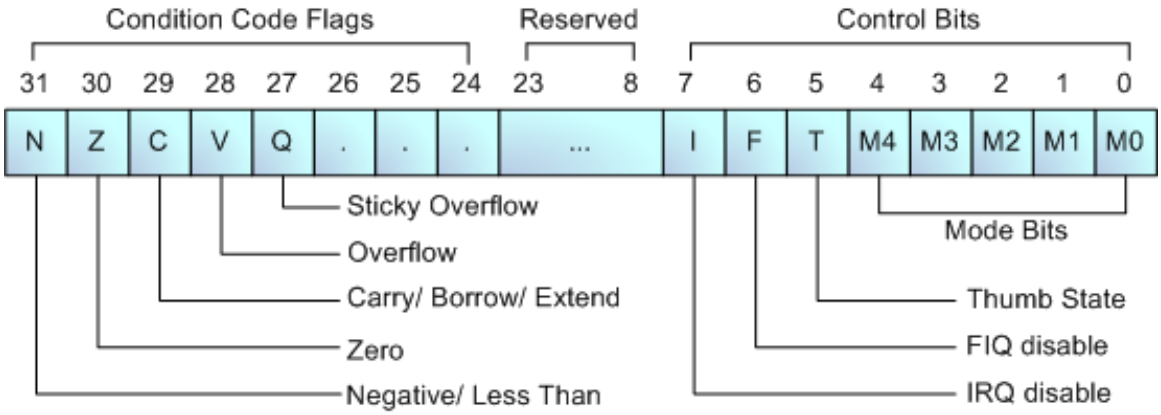
Modalità di funzionamento: le eccezioni (exceptions) 6

Il CPSR è il cuore della gestione delle eccezioni, oltre che a contenere i flags che descrivono il risultato dell'ultima operazione della ALU.

I modi di operazione sono conteggiati nei primi 5 bit.

Nei successivi 3 bit abbiamo la possibilità di disabilitare l'interrupt standard e la fast interrupt (FIQ).

L'ultimo bit, il bit T, svolge un ruolo importante nel tipo di funzionalità.



Se il bit T è settato si entra nella fase così detta Thumb.

L'ARM è un microcontrollore a 32 bit. Tutta questa potenzialità risulta ridondante se l'applicazione non richiede particolare sofisticazione in termini di prestazioni.

L'opzione Thumb consente di risparmiare risorse, fino al 30 %, semplicemente consentendo al microcontrollore di operare a 16 bit. Questa modalità è impostabile in un qualsiasi momento e posizione all'interno del programma.

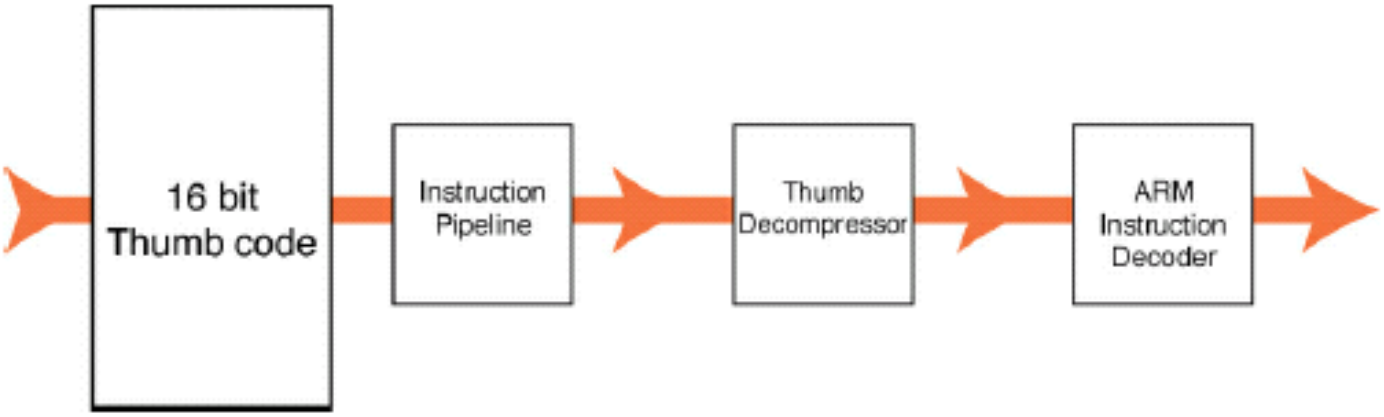
Il funzionamento in Thumb mode

Quando il funzionamento del micro viene commutato sul Thumb model il parallelismo diviene a 16 bit. Il set di istruzioni di cui si ha a disposizione nella modalità Thumb è ridotto.

L'occupazione di memoria si riduce, ed anche la complessità del codice.

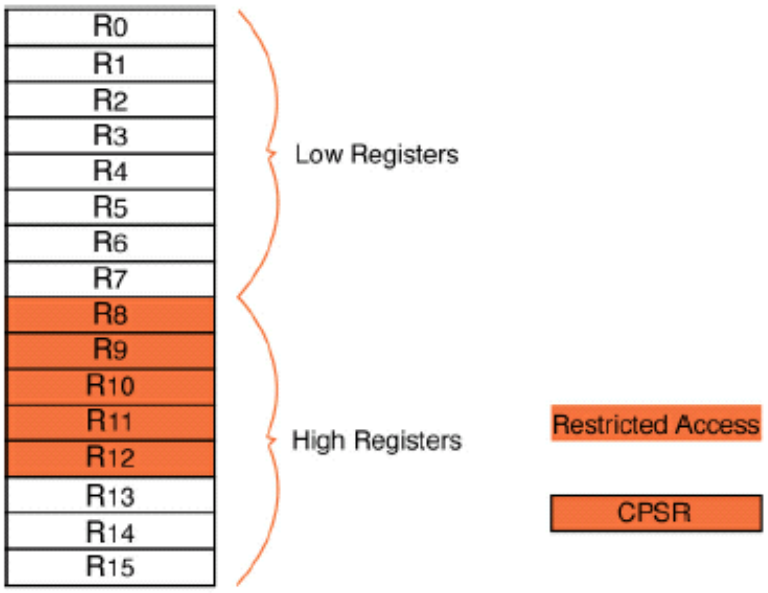
Ovviamente la velocità si riduce anch'essa.

Infatti l'organizzazione non è tale da possedere un sistema separato di decodifica delle istruzioni. Semplicemente le istruzioni vengono decompresse perché diventino di tipo ARM.



Una limitazione della modalità Thumb è nel numero di registri a disposizione: solo quelli da R0 a R7.

Un'altra restrizione riguarda le eccezioni, che non sono gestite in Thumb. Quando si verifica una eccezione si ha la conversione automatica da Thumb a ARM.



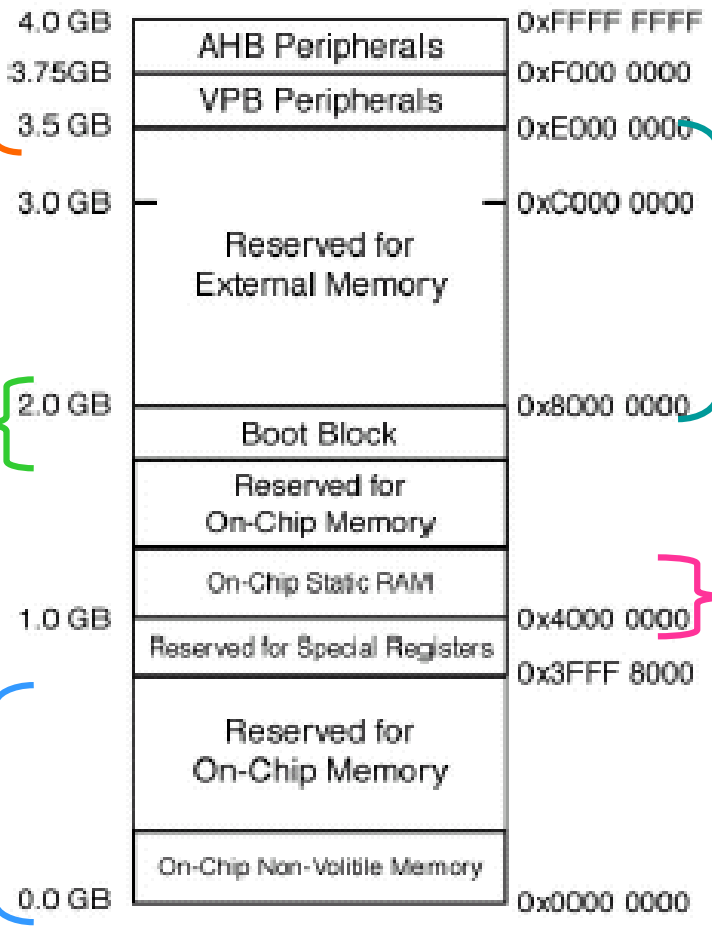
L'organizzazione della memoria 1

La memoria è organizzata come se fosse tutta impilata, indipendentemente dal tipo considerato. Questo significa che i vari blocchi di memoria vengono indirizzati con degli offset opportuni.

Tutti i registri di configurazione delle periferiche sono qui.

Il boot-loader program e debug program stanno qui.

Qui ci sta la on CHIP-FLASH, la cui dimensione dipende dal dispositivo.

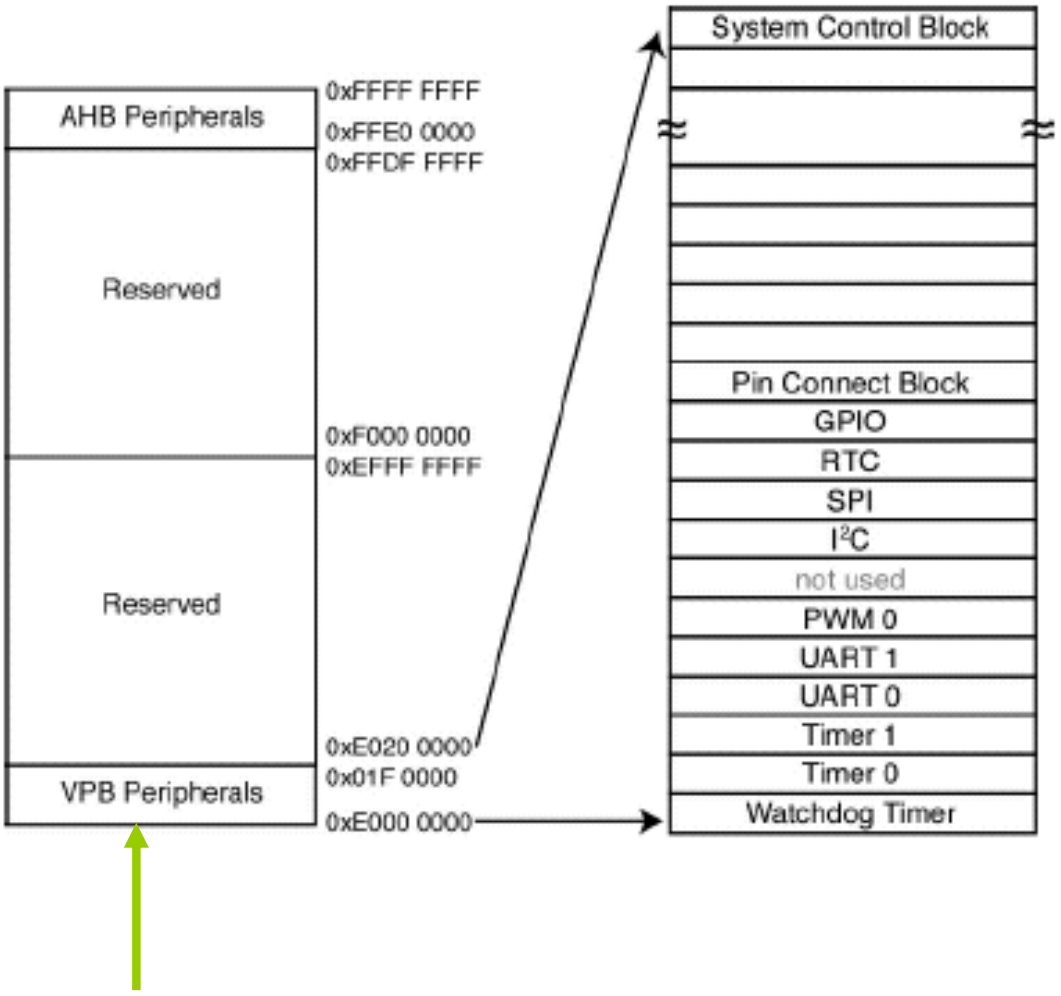


Una eventuale memoria programmi esterna sta qui.

La RAM è configurata dall'indirizzo 0x40000000.

L'organizzazione della memoria 2

Vediamo qui un'esplosione della parte di memoria riservata alla gestione delle periferiche (quella più in alto):



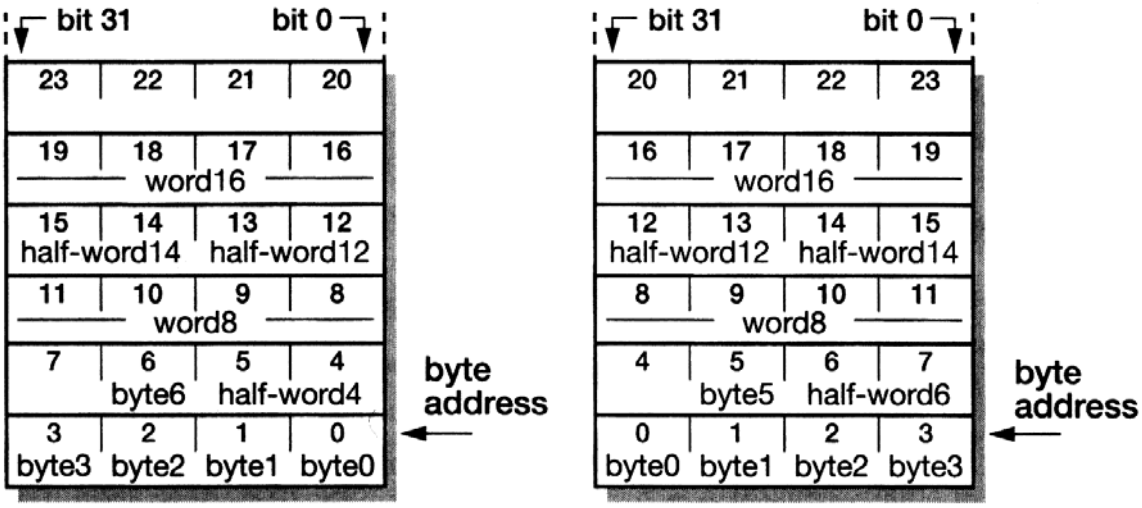
VPB sta per VLSI Peripheral bus,
ovvero tutte le periferiche legate all'HD.

L'organizzazione dei dati in memoria

L'organizzazione della memoria è a 32 bit. Però ogni parola di memoria è rappresentabile in multiplo di byte.

Va osservato che esistono 2 sistemi di rappresentazione del tutto equivalenti: **Little-Endian** e **Big-Endian**:

- **Little-Endian**: rappresentazione classica, il bit meno significativo sta a destra, quello più significativo a sinistra;
- **Big-endian**: il contrario del precedente, il bit più significativo sta a destra, quello meno significativo a sinistra.



(a) Little-endian memory organization *(b) Big-endian memory organization*

La parola a 32-bit è detta **Word**, quella a 16-bit, **Halfword**, quella a 8-bit, **Byte**..

Dal momento che si possono trattare dati di lunghezza minore di 32-bit occorre adottare un allineamento:

The ARM7TDMI processor supports the following data types:

- words, 32-bit
- halfwords, 16-bit
- bytes, 8-bit.

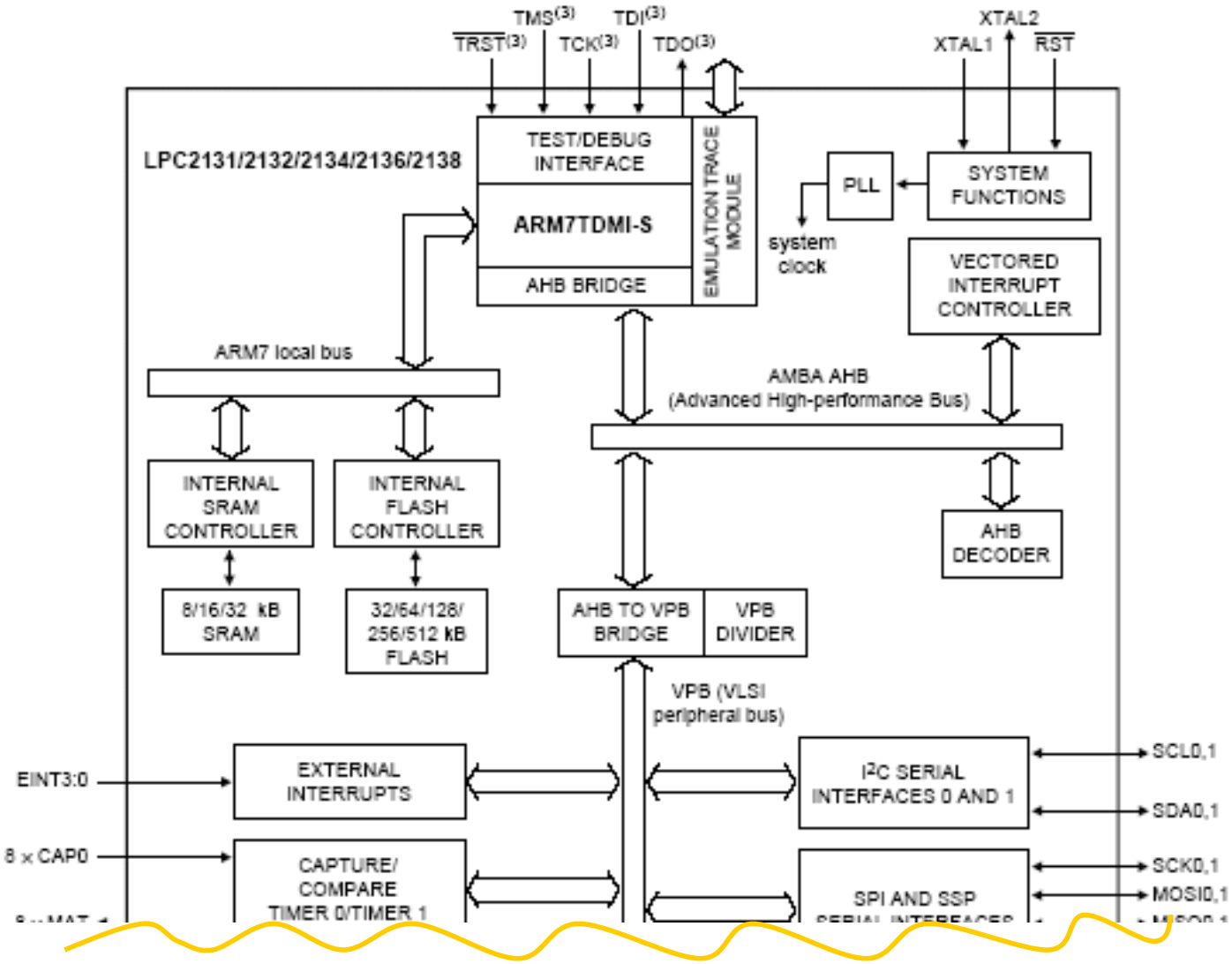
You must align these as follows:

- word quantities must be aligned to four-byte boundaries
- halfword quantities must be aligned to two-byte boundaries
- byte quantities can be placed on any byte boundary.

I BUS di comunicazione interna all'ARM 1

L'ARM è realizzato con un sistema di comunicazione dei dati interno ottimizzato in funzione della periferica. Questo bus si chiama **AMBA**, Advanced Microcontroller Bus Architecture, ed è specificato in 3 parti:

- Il Bus **AHB**, Advanced High-performance Bus per conettere periferiche ad alta velocità alla CPU;
- Il Bus **ASB**, Advanced System Bus, per connettere altri moduli ad alta performance;
- Il Bus **APB**, Advanced Peripheral Bus è per periferiche a bassa velocità.



I BUS di comunicazione interna all'ARM 2

La suddivisione del Bus di comunicazione secondo questa filosofia consente di potere trasferire i dati ad alta velocità verso le periferiche come la memoria ed allo stesso tempo gestire periferiche più lente come quelle dedicate alla trasmissione seriale con UART, etc.

Per mezzo di un 'bridge' si passa dal **AHB** al **APB**.

La particolarità importante dell'**APB** è che la sua velocità è impostabile come sottomultiplo della frequenza di clock.

In tale modo non si è forzati ad operare un unico bus alla frequenza più bassa, ma si hanno più bus operanti a velocità differente ed ottimizzata alla gestione della periferica.

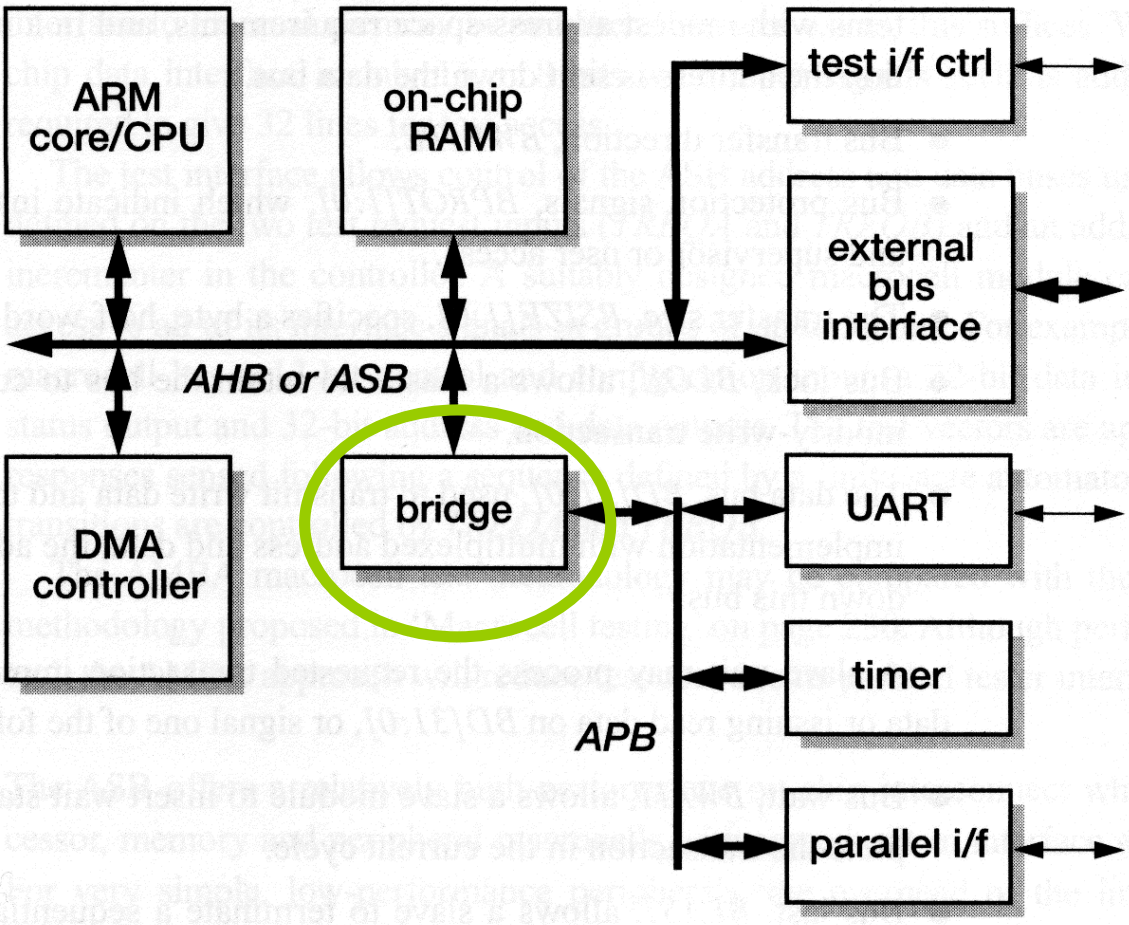


Figure 8.11 A typical AMBA-based system.

Qualche esempio applicativo

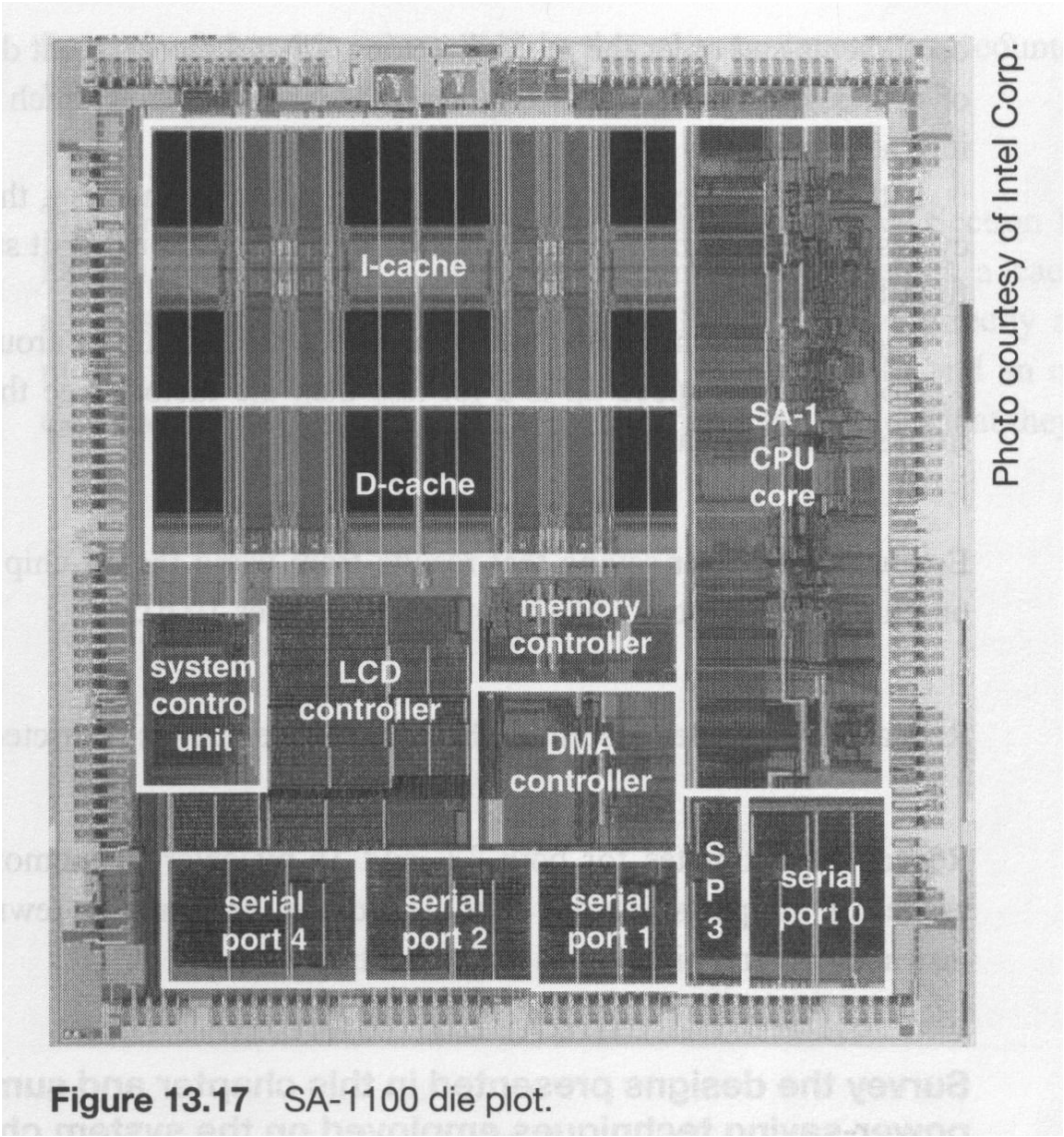


Figure 13.17 SA-1100 die plot.

Questo processore è realizzato con un processo avente:

Table 13.4 SA-1100 characteristics.

Process	0.35 µm	**Transistors**	2,500,000	**MIPS**	220/250
Metal layers	3	**Die area**	75 mm^2	**Power**	330/550 mW
Vdd	1.5/2 V	**Clock**	190/220 MHz	**MIPS/W**	665/450

Applicazioni

Sono notevoli le applicazioni con gli ARM. Visto che sono microcontrollori che riescono a realizzare sistemi aventi caratteristiche simili ai DSP:

Qui sotto abbiamo un tipico esempio di mobile phone handset:

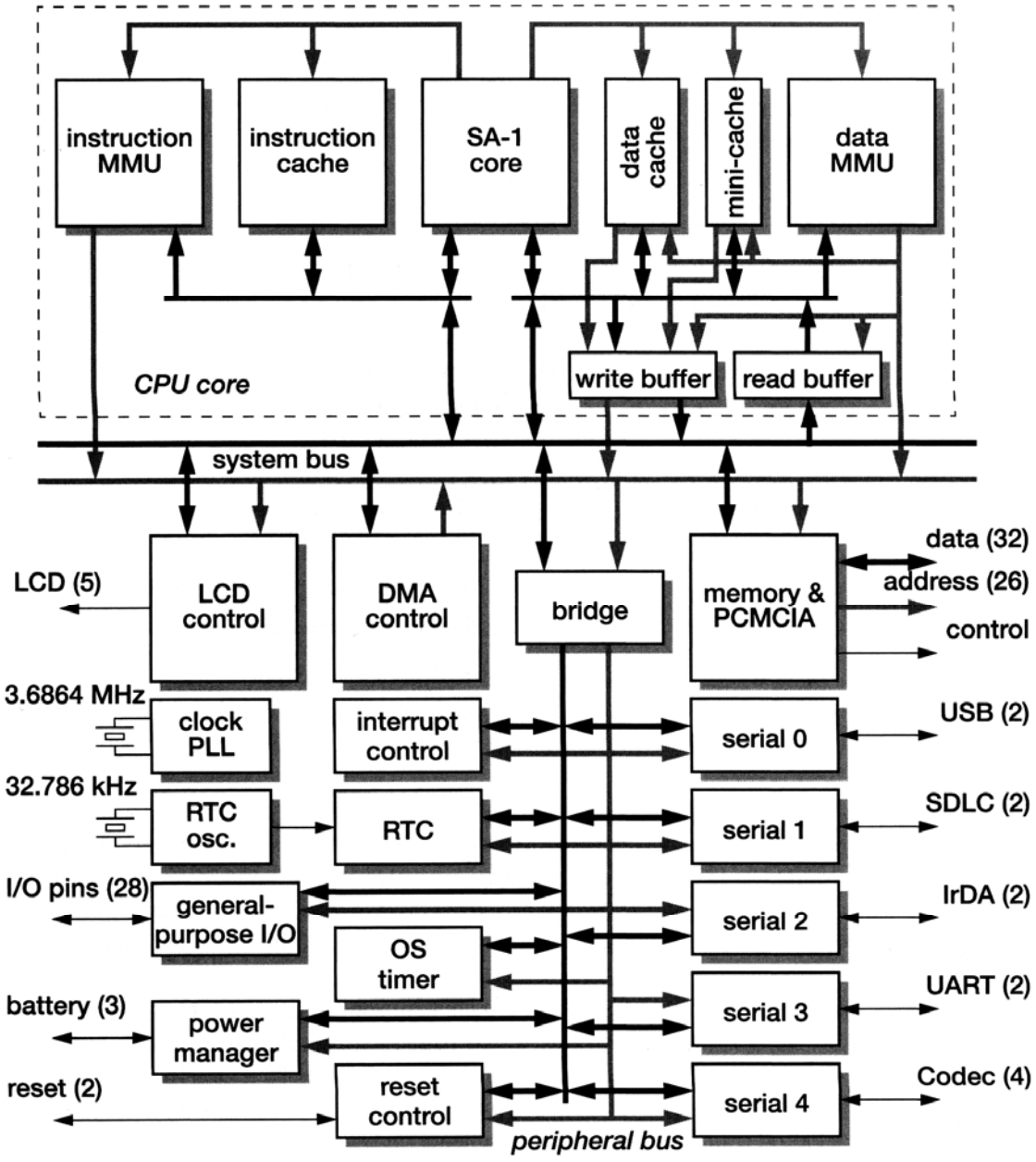


Figure 13.16 SA-1100 organization.

ARM - CORTEX M3 (1): introduzione (**)

I μ-controllori ARM sono stati sviluppati per cercare di soddisfare le emergenti richieste del mercato e sfruttare i benefici conseguibili dall'evoluzione tecnologica che consente di implementare sempre più funzioni.

D'altra parte ARM (Advanced RISC Machines ltd) stessa è una società così detta fables, vale a dire che sviluppa la struttura del processore e la parte fisica, l'implementazione elettrica del layout, senza realizzarla, ma vendendo la proprietà intellettuale in cambio di percentuali sui prodotti venduti. Per questa ragione sono molte le società che realizzano e vendono processori ARM.

Qui sotto abbiamo la tabella che mostra le famiglie di processori. L'ordine è per famiglia, non per potenza elaborativa. ARM ha sviluppato le varie famiglie in funzione delle esigenze di mercato.

Table 1.1 ARM Processor Names

Processor Name	Architecture Version	Memory Management Features	Other Features
ARM7TDMI	ARMv4T		
ARM7TDMI-S	ARMv4T		
ARM7EJ-S	ARMv5E		DSP, Jazelle
ARM920T	ARMv4T	MMU	
ARM922T	ARMv4T	MMU	
ARM926EJ-S	ARMv5E	MMU	DSP, Jazelle
ARM946E-S	ARMv5E	MPU	DSP
ARM966E-S	ARMv5E	DSP	
ARM968E-S	ARMv5E		DMA, DSP
ARM966HS	ARMv5E	MPU (optional)	DSP
ARM1020E	ARMv5E	MMU	DSP
ARM1022E	ARMv5E	MMU	DSP
ARM1026EJ-S	ARMv5E	MMU or MPU	DSP, Jazelle
ARM1136J(F)-S	ARMv6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	ARMv6	MMU + TrustZone	DSP, Jazelle
ARM11 MPCore	ARMv6	MMU + multiprocessor cache support	DSP, Jazelle
ARM1156T2(F)-S	ARMv6	MPU	DSP
Cortex-M0	ARMv6-M		NVIC
Cortex-M1	ARMv6-M	FPGA TCM interface	NVIC
Cortex-M3	ARMv7-M	MPU (optional)	NVIC

La tabella continua → → →

(**): un grazie a Filippo Bianchi e Lorenzo Rota Per gli importanti dettagli suggeriti.

ARM - CORTEX M3 (2)

Una significativa modifica al cuore del processore si è avuta con la generazione dei CORTEX.

Table 1.1 ARM Processor Names *Continued*

Processor Name	Architecture Version	Memory Management Features	Other Features
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP + Floating point
Cortex-A8	ARMv7-A	MMU + TrustZone	DSP, Jazelle, NEON + floating point
Cortex-A9	ARMv7-A	MMU + TrustZone + multiprocessor	DSP, Jazelle, NEON + floating point

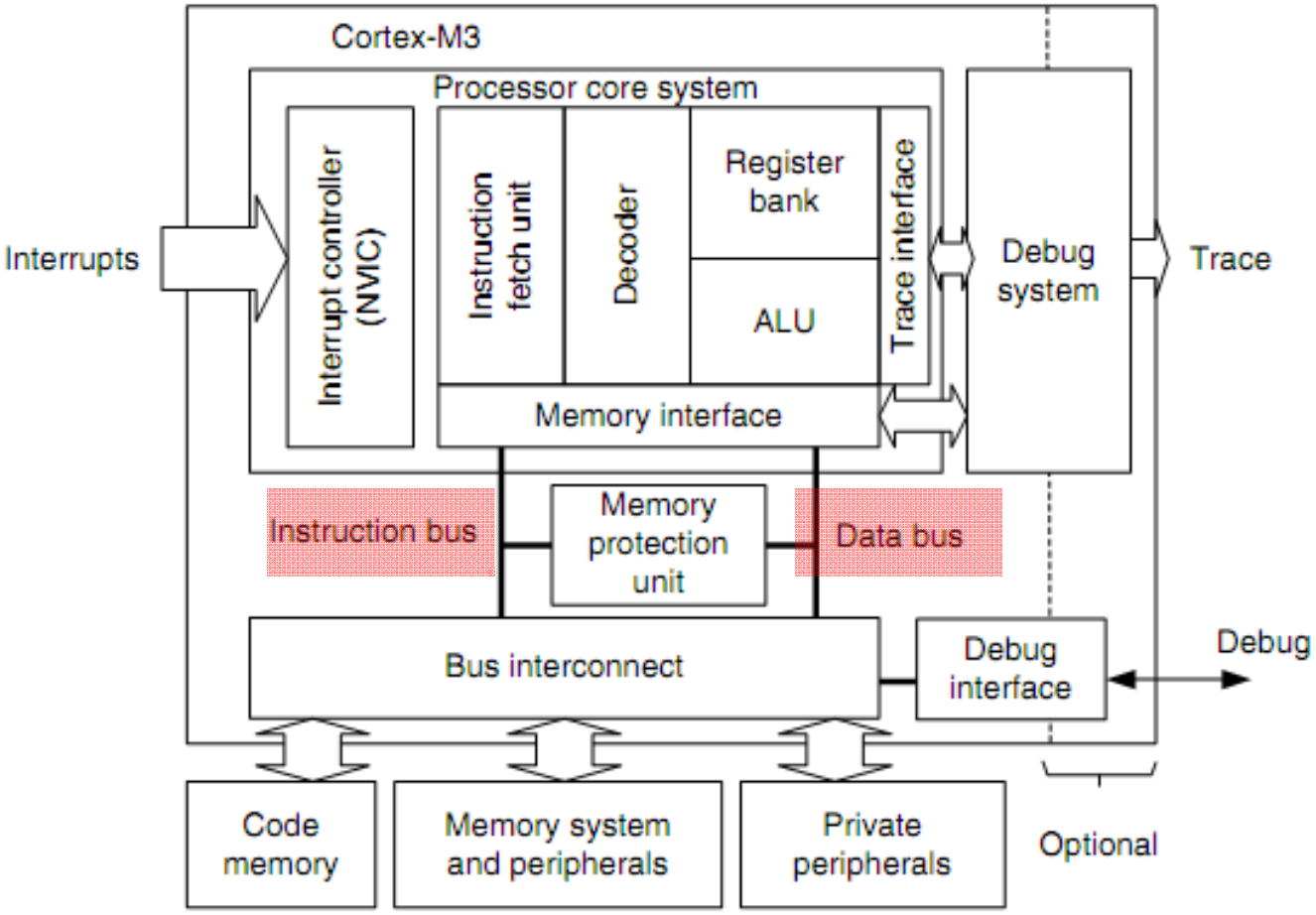
Questa generazione di microprocessori rispondono alle nuove esigenze di mercato in termini di portabilità e gestione di molti processi di comunicazione.

Le nuove generazioni di ARM sono stati adottate negli smart-phones e nei tablet PC con ottimi risultati di mercato.

Le innovazioni stanno nella velocità di esecuzione del codice ma anche nel basso risparmio energetico di fondamentale importanza nelle applicazioni a batteria.

ARM - CORTEX M3 (3): unità centrale

La struttura interna di un Cortex presenta significative differenze rispetto a quella del tradizionale ARM.



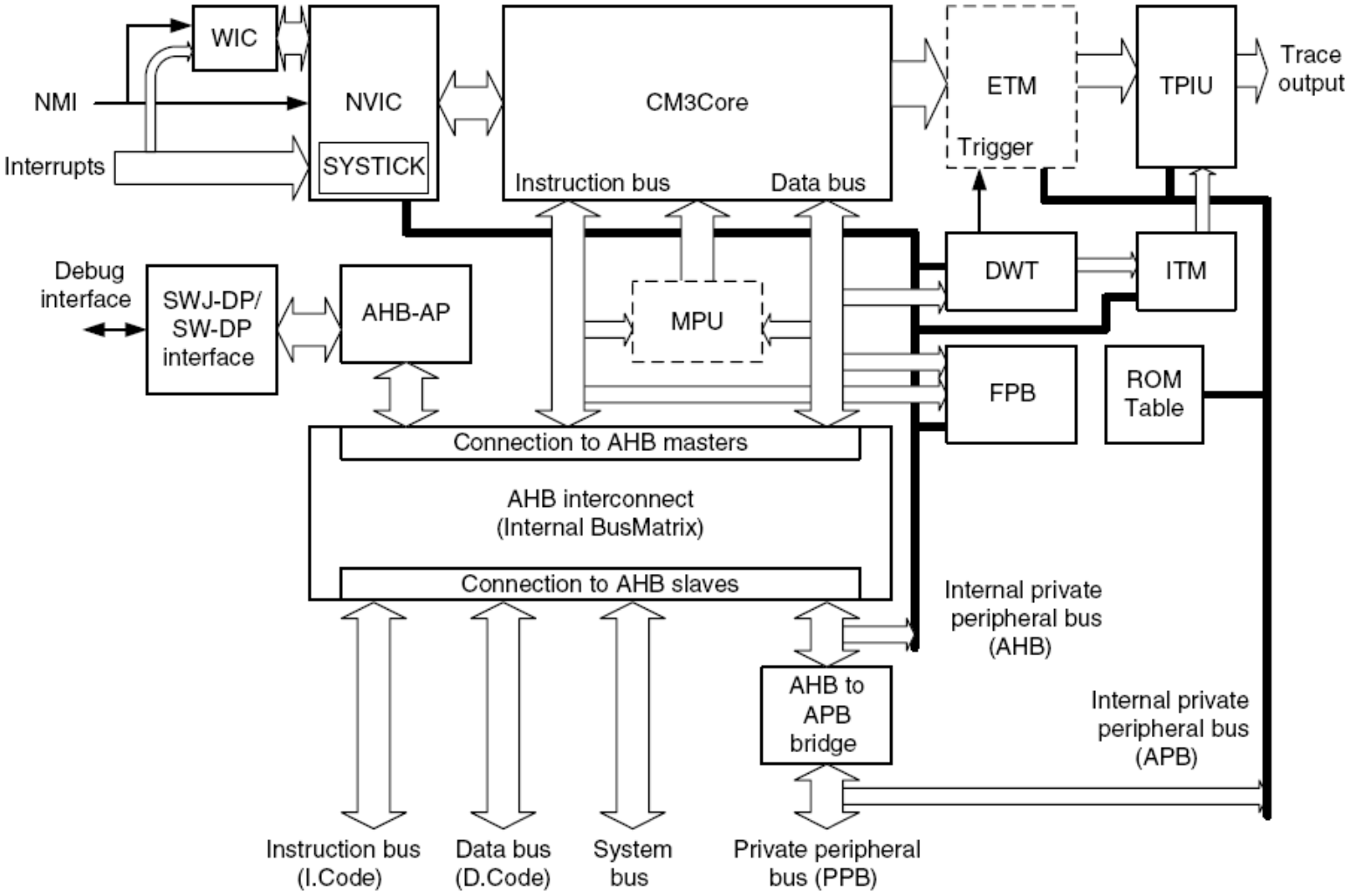
Nel Cortex all'interno della CPU ora si hanno il bus istruzioni ed il bus dati separati: la struttura è diventata puramente Harvard. Questo consente un significativo miglioramento nelle prestazioni in velocità ed un'ottimizzazione della struttura pipeline.

La separazione dei 2 bus consente una razionalizzazione del funzionamento. Tuttavia la memoria viene considerata come se fosse una sola, anche se fisicamente sono distinte. Intervalli di indirizzamento stabiliti identificano e caratterizzano il tipo di memoria che si sta trattando.

Ovviamente la locazione utilizzata è trasparente all'utilizzatore e viene assegnata direttamente dal compilatore se si usano i soliti criteri di definizione in C (nella flash viene messo il codice e i dati dichiarati costanti).

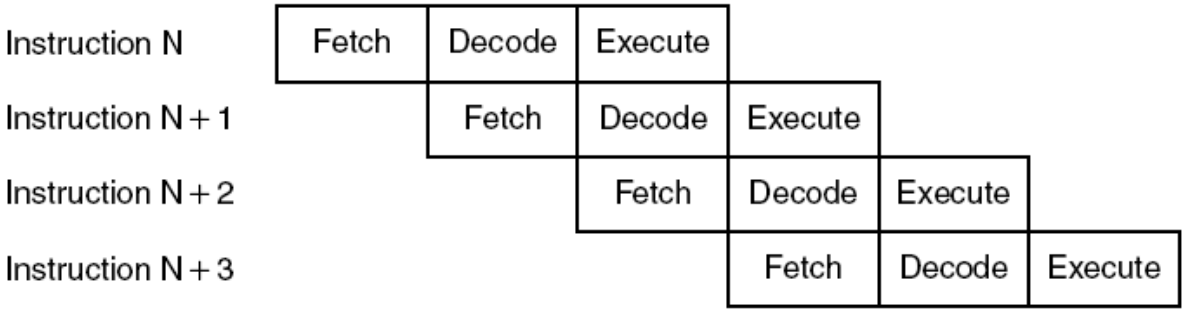
ARM - CORTEX M3 (4)

Qui abbiamo una visione più aperta dei blocchi principali del CORTEX. Dove si vedono tutti i blocchi che servono al debug sulla destra.



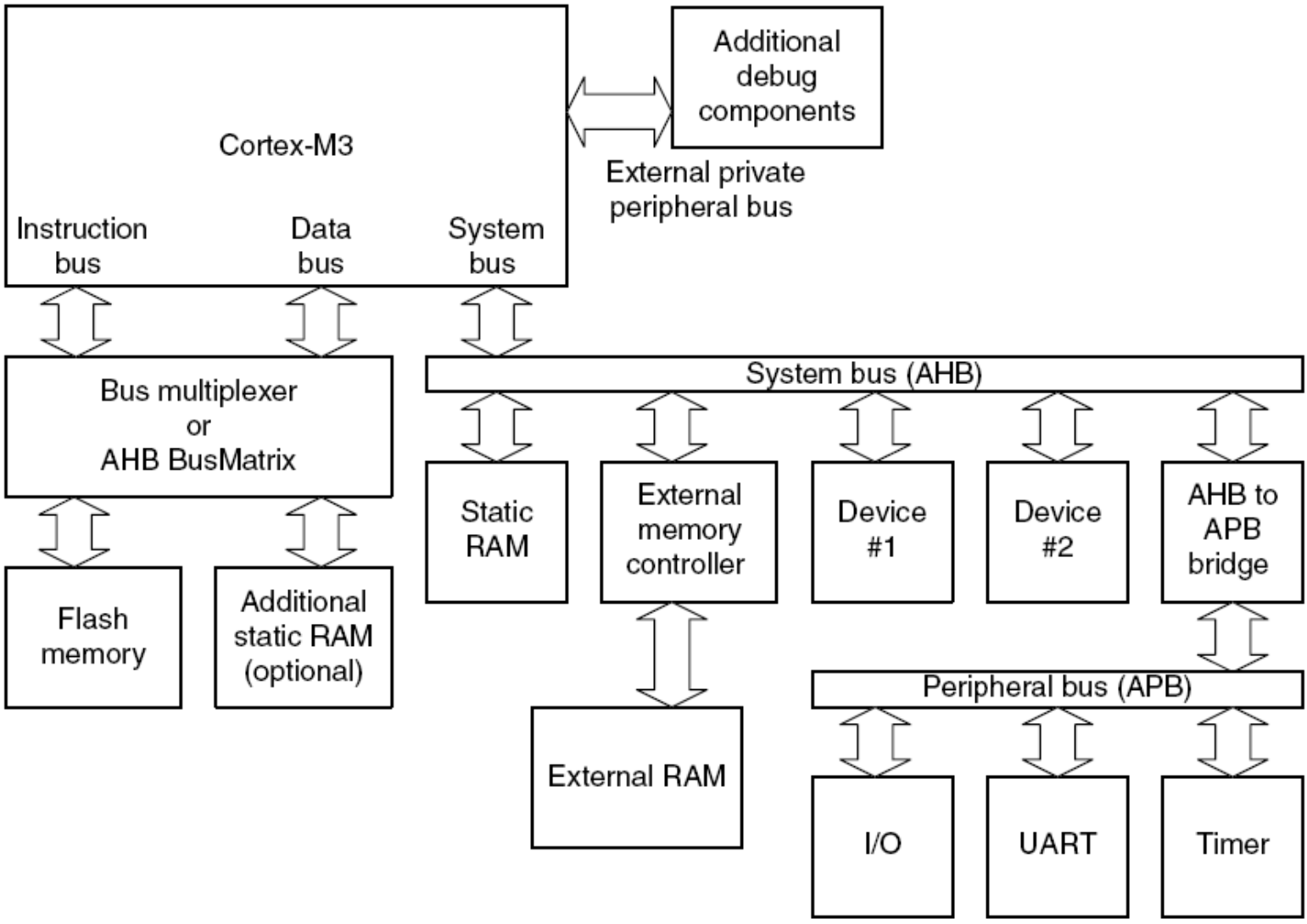
C'e' un blocco dedicato alla gestione delle interruzioni. Si tratta del NVIC (Nested Vectored Interrupt Controller). In questo blocco risiede anche il Systick, l'orologio utilizzato dal sistema operativo, quando adottato nell'applicazione.

Il CORTEX ha una profondità di pipeline di 3 stadi:



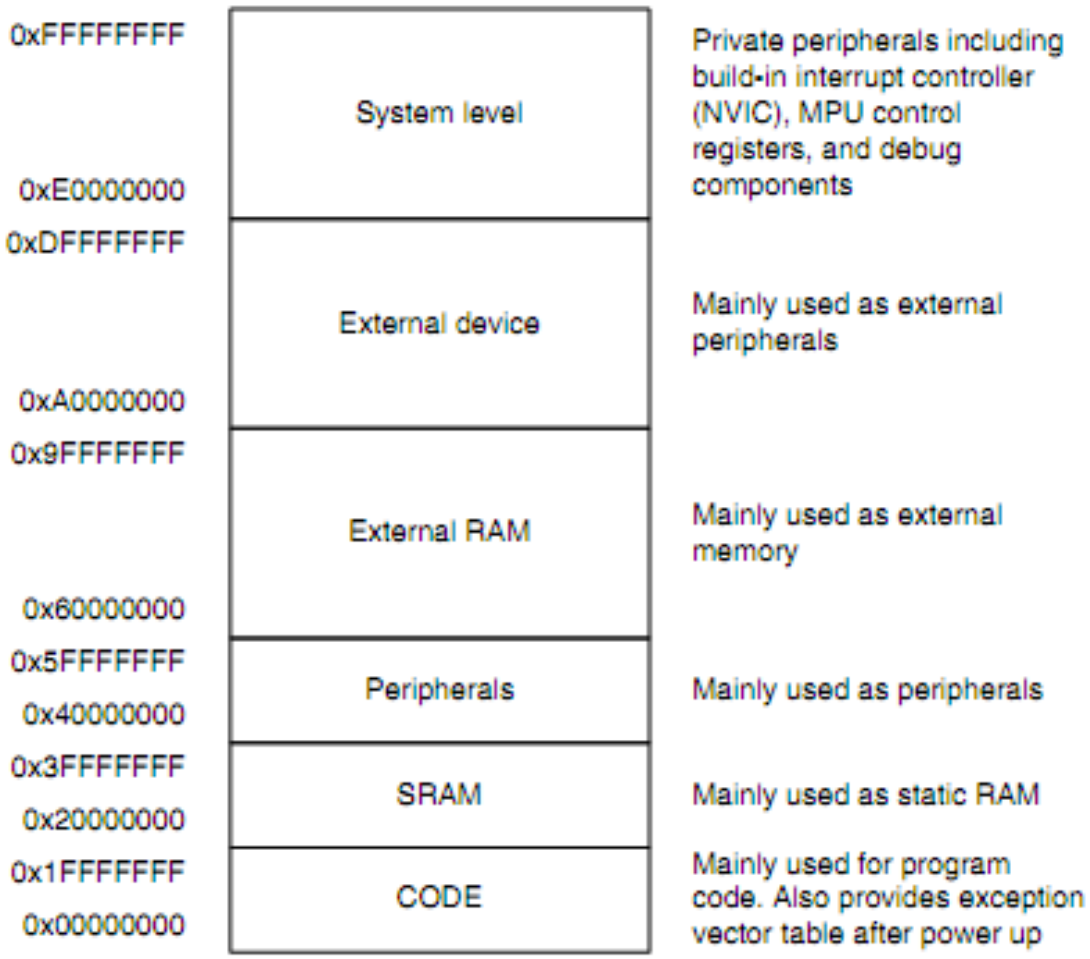
ARM - CORTEX M3 (5)

In questo altro diagramma espanso sono messi invece in evidenza tutti i blocchi interni al micro con i bus di interfaccia usati in funzione delle periferiche.



ARM - CORTEX M3 (6): memoria

Ecco come si evidenzia la mappa della memoria. Fisicamente i blocchi possono essere memorie differenti i cui indirizzi devono essere codificati come sotto riportati per potere essere individuati ed interpretati.

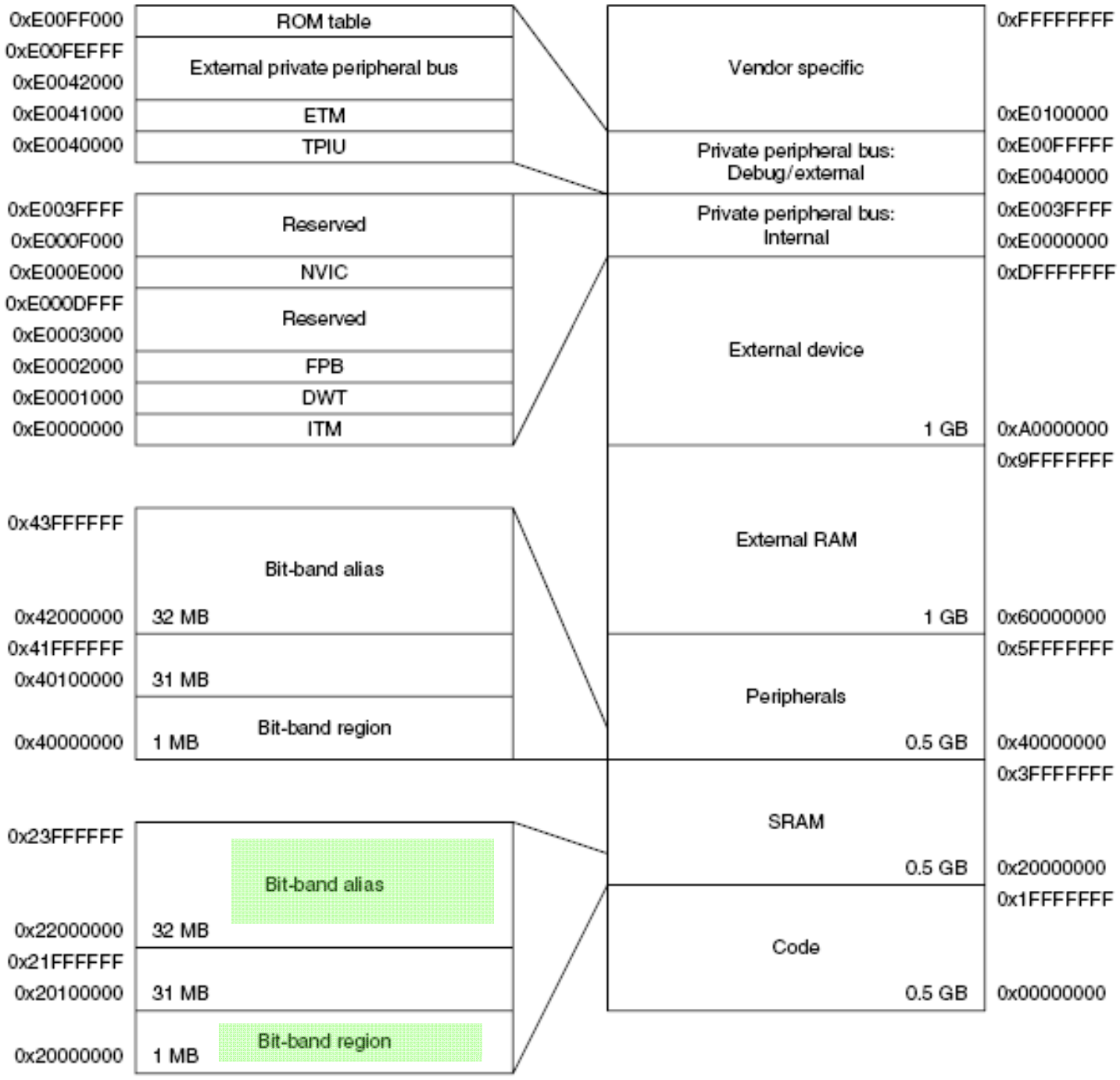


Al solito la memoria programmi parte dall'indirizzo 0, zona in cui risiedono anche i vettori per le interruzioni.

Oramai la memoria integrata riesce ad avere capacità adeguata. Quindi la memoria programmi risiede o tutta o la prima parte all'interno del micro.

ARM - CORTEX M3 (7)

Ecco un esempio di distribuzione della memoria più in dettaglio:



Di nuovo nel Cortex c'e' la regione di bit-band dove si può adottare l'indirizzamento per bit.

Sostanzialmente, all'indirizzo 0x20yyyyyy ci sono le celle che possono essere memorizzate per bit o per parole. Per scrivere il bit si usa la memoria alias di indirizzo 0x22yyyyyy. In sostanza ogni cella nella memoria alias contiene solo 1 bit (gli altri 31 vengono buttati). Ecco la ragione per cui se a 0x20yyyyyy c'è 1 Mb di spazio, a 0x22yyyyyy ne servono 32 Mb.

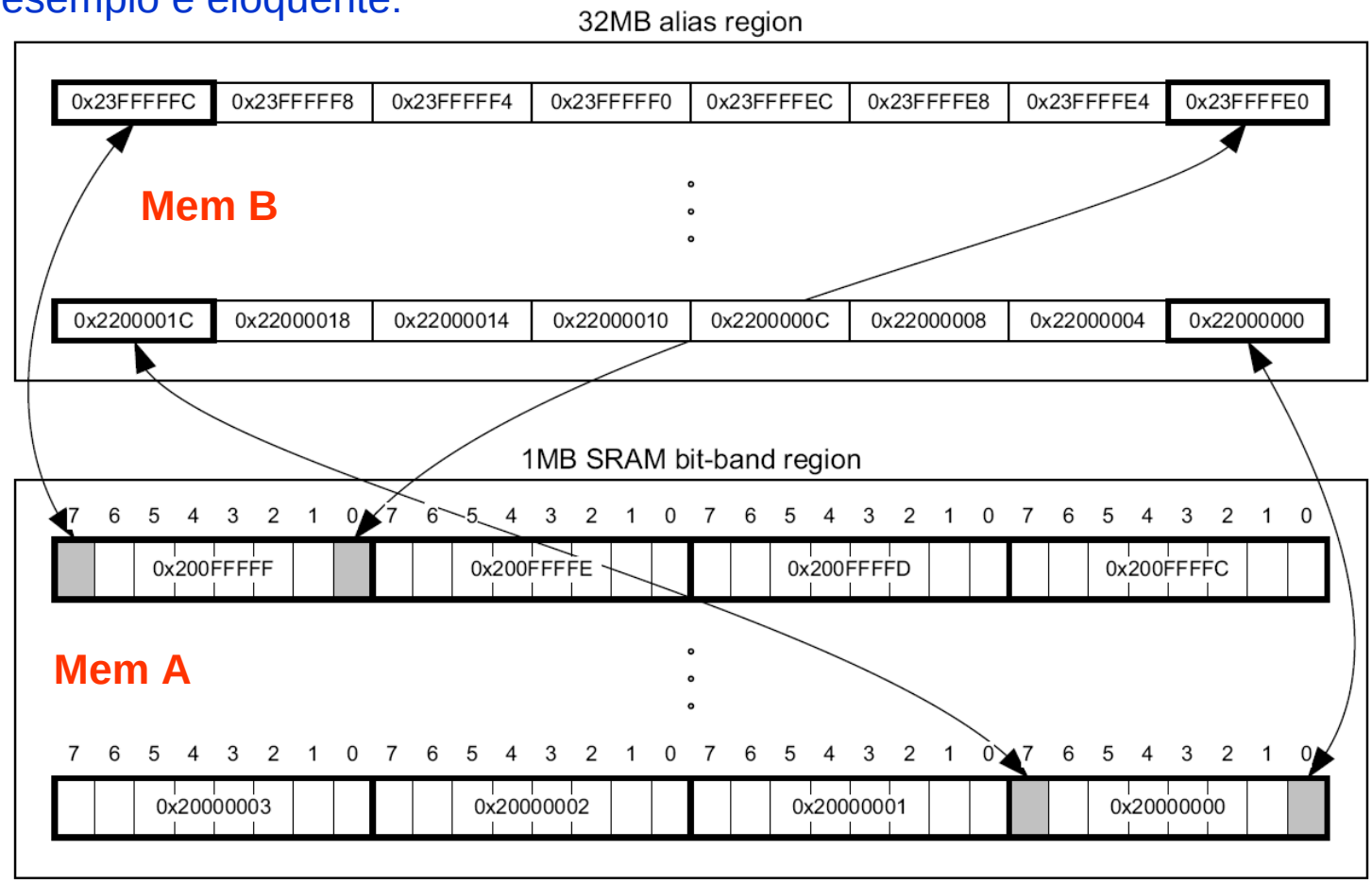
ARM - CORTEX M3 (8): bit-band

Anche nell'8051 abbiamo visto la presenza di una zona di memoria indirizzabile per parola e per bit.

La regione di memoria dove sono i registri bit-band interpretabili come registri standard inizia con: 0x20yyyyyy.

Se si intende indirizzare i singoli bit si deve usare l'alias 0x22yyyyyy.
L'indirizzo di ogni bit si ottiene incrementando per 4, partendo da 0.

L'esempio è eloquente:



Il metodo consiste nel ri-mappare un'area di memoria A in un'area di memoria B, detta Bit Band Aalias. Ad ogni bit della memoria A corrisponde una word (32 bit) della memoria B. Andando a scrivere un valore diverso da zero in una word di B, il corrispondente bit nella memoria A viene posto ad 1. Da qui la ragione di disporre di una memoria di alias che sia 32 volte più grande della memoria di bit-mapping. La formula per l'alias è:

$$Ind\_in\_B = Offset\_mem\_B + \left(Ind\_word\_del\_bi\_in\_A - Offset\_mem\_A\right) * 32 + n\_bit * 4$$

ARM - CORTEX M3 (9): bit-band

In Assembly l'implementazione del bit-mapping è ovvia.

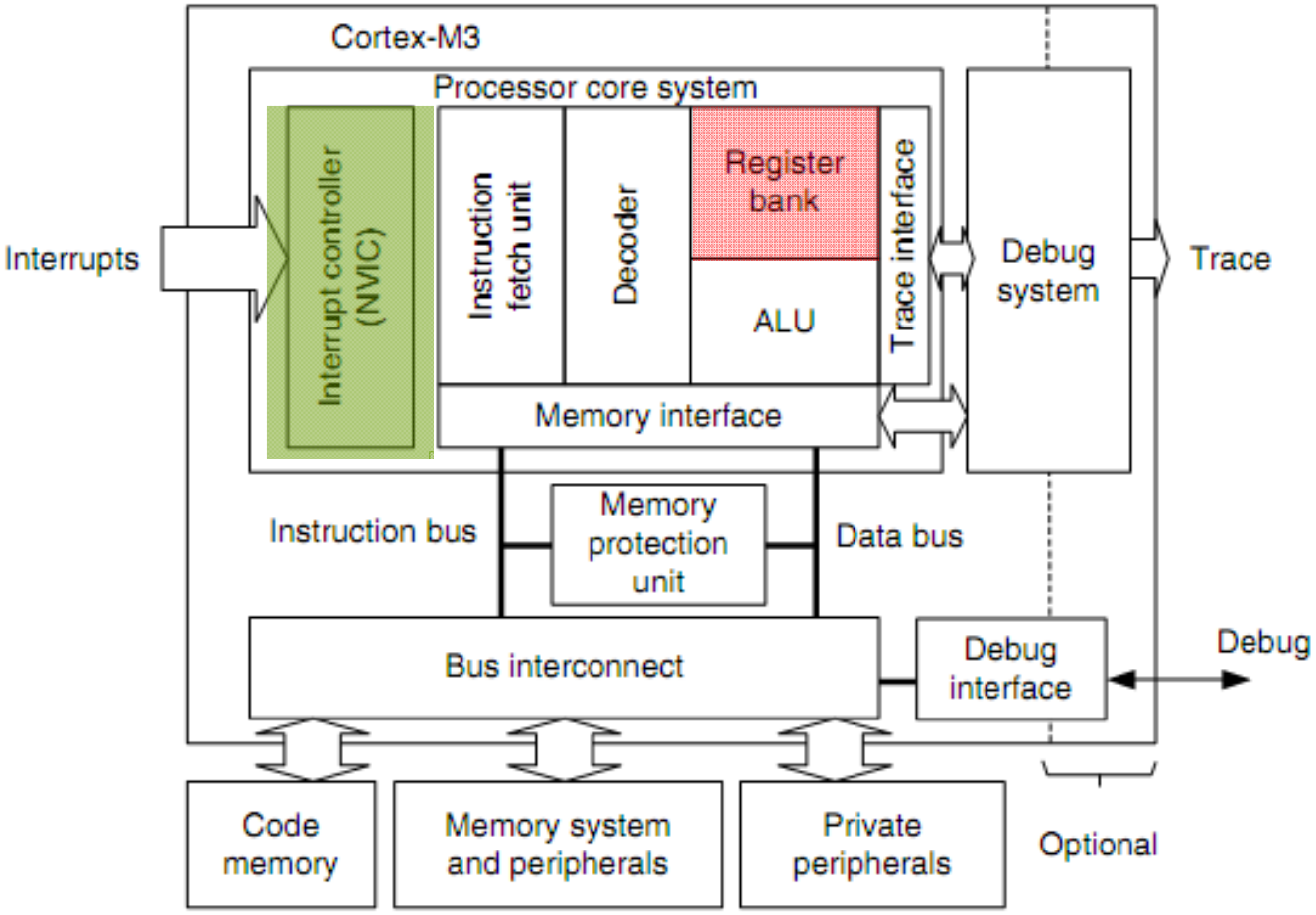
Siccome in C non esiste il corrispettivo si deve introdurre l'alias:

```
#define  DEVICE_REG0      *((volatile unsigned long *)  (0x40000000))
#define  DEVICE_REG0_BIT0 *((volatile unsigned long *)  (0x42000000))
#define  DEVICE_REG0_BIT1 *((volatile unsigned long *)  (0x42000004))
...
DEVICE_REG0 = 0xAB; // Accessing the hardware register by normal
                    // address
...
DEVICE_REG0 = DEVICE_REG0 | 0x2; // Setting bit 1 without using
                                 // bitband feature
...
DEVICE_REG0_BIT1 = 0x1; // Setting bit 1 using bitband feature
                        // via the bit band alias address
```

Informazioni su questa proprietà e su molte altre proprietà del Cortex, nonché della manualistica si possono trovare nei siti: www.ARM.com e www.OnArm.com.

ARM - CORTEX M3 (10): registri

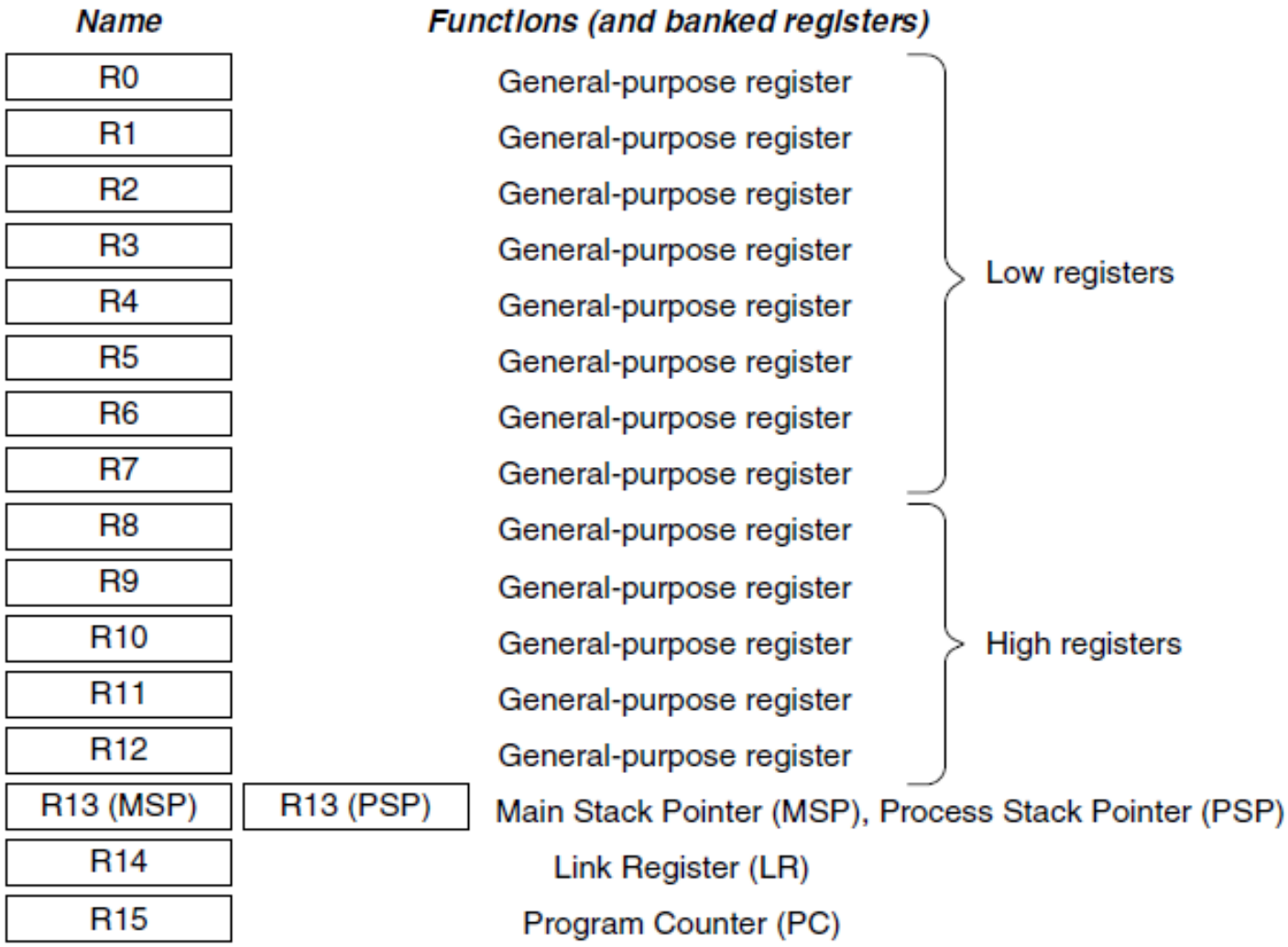
Continuiamo nella verifica della differente natura del Cortex e del classico ARM.



Una differenza peculiare tra l'ARM tradizionale ed il Cortex è la capacità di gestire Interruzioni multiple in naturale mediante il blocco NVIC, che sta appunto per Nested Vector Interrupt Controller. Come vedremo tra breve. In particolare l'NVIC fa parte dell'unità centrale. Mentre nell'ARM7 era sviluppato in proprio da chi implementava il micro stesso.

La possibilità di gestire interruzioni temporanee ha portato ad una diversa distribuzione del banco di registri interni alla CPU.

ARM - CORTEX M3 (11)



I banchi di registri che venivano scambiati all'occorrenza delle interruzioni, in funzione del tipo di interruzione sono spariti tutti. E' rimasto solo un banco di 16 registri di cui i primi 13 sono di utilizzo generale.

R13 è lo stack pointer, rappresentato da 2 registri. L'MSP è il solo usato durante l' handler mode (interruzioni). PSP o MSP possono essere usati nel thread mode (modalità standard). La selezione è effettuabile nel Control Register.

R14 è il classico Link Register dove va ad essere memorizzato il contenuto del Program Counter quando viene chiamato un sottoprogramma di qualsiasi natura.

Infine R15 è il Program Counter vero e proprio, il cui valore può essere modificato dall'utilizzatore se si intendono eseguire salti di tipo particolare. Alcune di queste modifiche possono essere eseguite solo in Assembly, altre sono visibili anche al C .

ARM - CORTEX M3 (12)

Accanto ai registri di utilizzo generale ci sono gli Special Registers dove sono contenute molte informazioni provenienti dai vari organi che compongono la CPU.

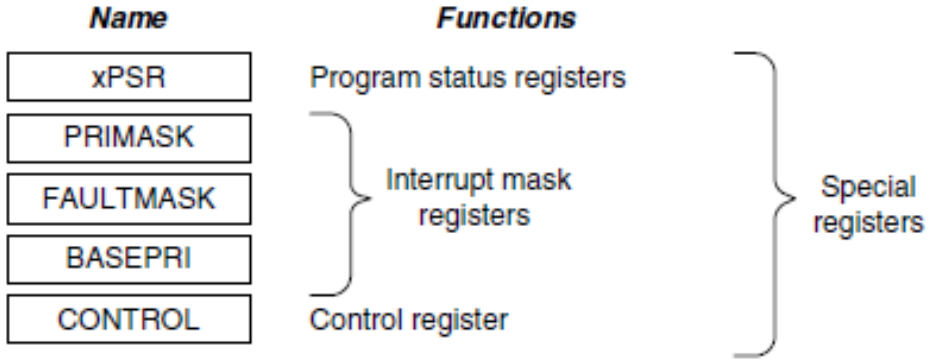


FIGURE 2.3

Special Registers in the Cortex-M3.

Table 2.1 Special Registers and Their Functions

Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

For more information on these registers, see Chapter 3.

L' xPSR è l'OR di 3 registri riempiti in modo complementare:

	31	*30*	*29*	*28*	*27*	*26:25*	*24*	*23:20*	*19:16*	*15:10*	*9*	*8*	*7*	*6*	*5*	*4:0*
APSR	N	Z	C	V	Q											
IPSR												Exception number				
EPSR						ICI/IT	T			ICI/IT						

Le definizioni dei 3 registri:

- Application Program Status register (APSR)
- Interrupt Program Status register (IPSR)
- Execution Program Status register (EPSR)

ARM - CORTEX M3 (13)

Siccome i 3 registri precedenti sono riempiti in posizioni complementari se ne può considerare l'OR, ottenendo il vero e proprio xPSR:

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception number				

Il significato dei veri flags è:

Table 3.1 Bit Fields in Cortex-M3 Program Status Registers

Bit	Description
N	Negative
Z	Zero
C	Carry/borrow
V	Overflow
Q	Sticky saturation flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit
T	Thumb state, always 1; trying to clear this bit will cause a fault exception
Exception number	Indicates which exception the processor is handling

I 6 bit più significativi sono riferiti al comportamento della ALU, ovvero al risultato dell'ultima operazione matematica.

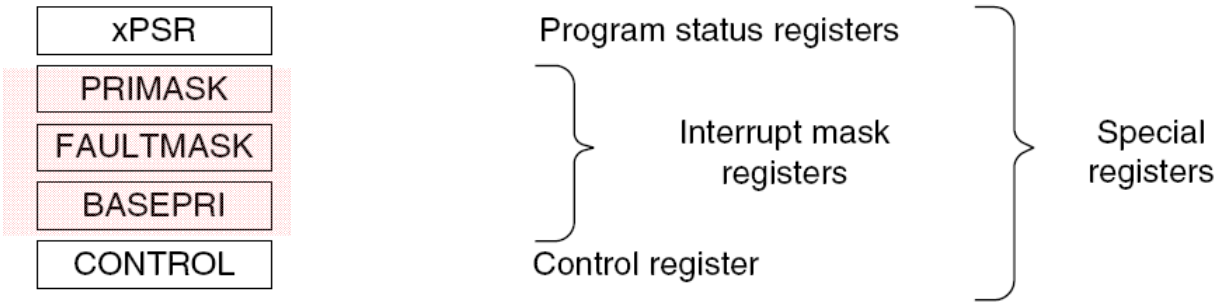
ICI/IT è il flag che tiene conto del fatto che ora le interruzioni multiple sono gestite direttamente dal micro.

T è il bit di thumb-mode che si differenzia dall'ARM classico e che rimane sempre a 1.

Exception Number è che interruzione si sta gestendo. Anche qui c'è distinzione dall'ARM classico dove qui era invece predisposta la modalità di funzionamento.

ARM - CORTEX M3 (14)

I registri PRIMASK, FAULTMASK e BASEPRI sono tutti e 3 legati alle interruzioni.



I primi 2 registri sono registri a singolo bit (1-bit register) che tengono conto della condizioni di mascherabilità o non-mascherabilità di un'interruzione.

Table 3.2 Cortex-M3 Interrupt Mask Registers

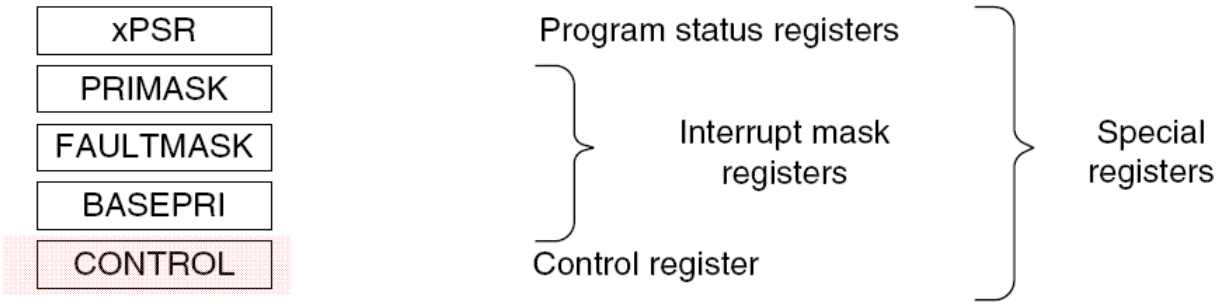
Register Name	Description
PRIMASK	A 1-bit register, when this is set, it allows nonmaskable interrupt (NMI) and the hard fault exception; all other interrupts and exceptions are masked. The default value is 0, which means that no masking is set.
FAULTMASK	A 1-bit register, when this is set, it allows only the NMI, and all interrupts and fault handling exceptions are disabled. The default value is 0, which means that no masking is set.
BASEPRI	A register of up to 8 bits (depending on the bit width implemented for priority level). It defines the masking priority level. When this is set, it disables all interrupts of the same or lower level (larger priority value). Higher priority interrupts can still be allowed. If this is set to 0, the masking function is disabled (this is the default).

Il terzo registro, BASEPRI, è a 8-bit e tiene conto delle priorità. Quando impostato mette in attesa tutte le priorità minori o uguali a quella che si sta eseguendo.

Se invece una interruzione di priorità superiore si verifica può prendere il sopravvento.

ARM - CORTEX M3 (15)

Ed infine abbiamo il CONTROL Register.



Il CONTROL Register ha solo 2 bit utilizzati che hanno il significato descritto qui sotto:

Table 3.3 Cortex-M3 Control Register

Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be 0 when the processor is in handler mode.
CONTROL[0]	0 = Privileged in thread mode 1 = User state in thread mode If in handler mode (not thread mode), the processor operates in privileged mode.

ARM - CORTEX M3 (16): interruzioni

Le interruzioni non sono più suddivise in FAST, SWI, ecc come nell'ARM. Ma sono semplicemente numerate. La priorità che si imposta determina il grado di precedenza.

Ci sono delle eccezioni, o interruzioni, che sono comuni a tutti i micro, poi ci sono quelle individuabili ed ammesse all'utilizzatore. Nella tabella sotto sono 240.

La priorità è tanto maggiore quanto più piccolo è il valore assegnato. Per tale ragione al reset è assegnata la priorità maggiore, -3, cui segue l'interruzione non-mascherata, ovvero che può verificarsi senza essere subordinata ad altre eventualità con priorità -2. Quindi ci sono una serie di condizioni di interruzione dovuti ad errori che possono verificarsi.

Table 3.4 Exception Types in Cortex-M3

Exception Number	Exception Type	Priority	Function
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt
3	Hard fault	–1	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking
4	MemManage	Settable	Memory management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a nonexecutable region)
5	Bus fault	Settable	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Settable	Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3)
7–10	—	—	Reserved
11	SVC	Settable	Supervisor call via SVC instruction
12	Debug monitor	Settable	Debug monitor
13	—	—	Reserved
14	PendSV	Settable	Pendable request for system service
15	SYSTICK	Settable	System tick timer
16–255	IRQ	Settable	IRQ input #0–239

ARM - CORTEX M3 (17)

Al solito al verificarsi dell'interruzione deve essere chiamato il sottoprogramma che la gestisce. Per questo è adibita una tabella di vettori, o puntatori, dove sono contenuti gli indirizzi dei sottoprogrammi a cui si deve saltare.

Table 3.5 Vector Table Definition after Reset

Exception Type	Address Offset	Exception Vector
18–255	0x48–0x3FF	IRQ #2–239
17	0x44	IRQ #1
16	0x40	IRQ #0
15	0x3C	SYSTICK
14	0x38	PendSV
13	0x34	Reserved
12	0x30	Debug monitor
11	0x2C	SVC
7–10	0x1C–0x28	Reserved
6	0x18	Usage fault
5	0x14	Bus fault
4	0x10	MemManage fault
3	0x0C	Hard fault
2	0x08	NMI
1	0x04	Reset
0	0x00	Starting value of the MSP

La tabella dei vettori è rilocabile. Infatti sopra qui sono individuati gli offset da aggiungere all'indirizzo di partenza della tabella.

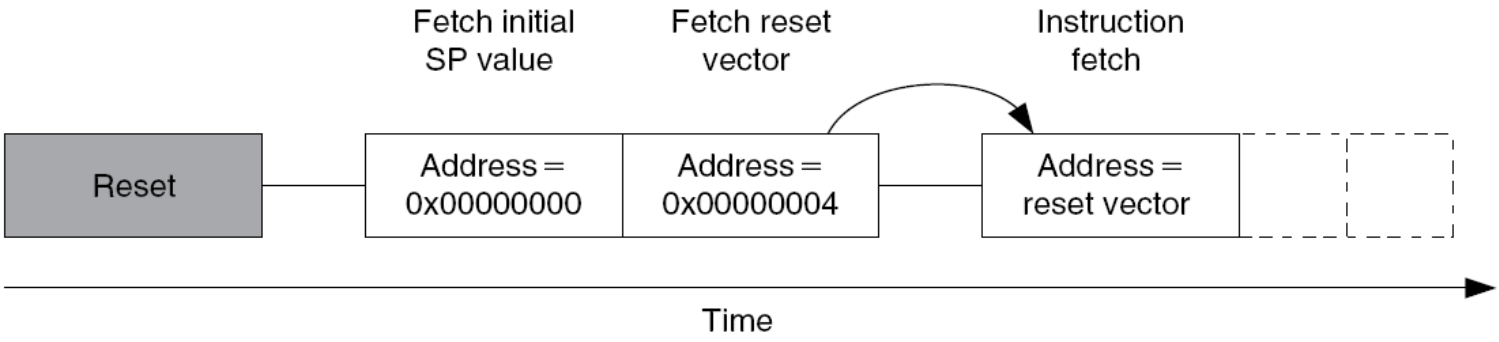
L'indirizzo di partenza della tabella è contenuto in uno dei registri NVIC, precisamente il Vector Table Offset Register, con questa sintassi:

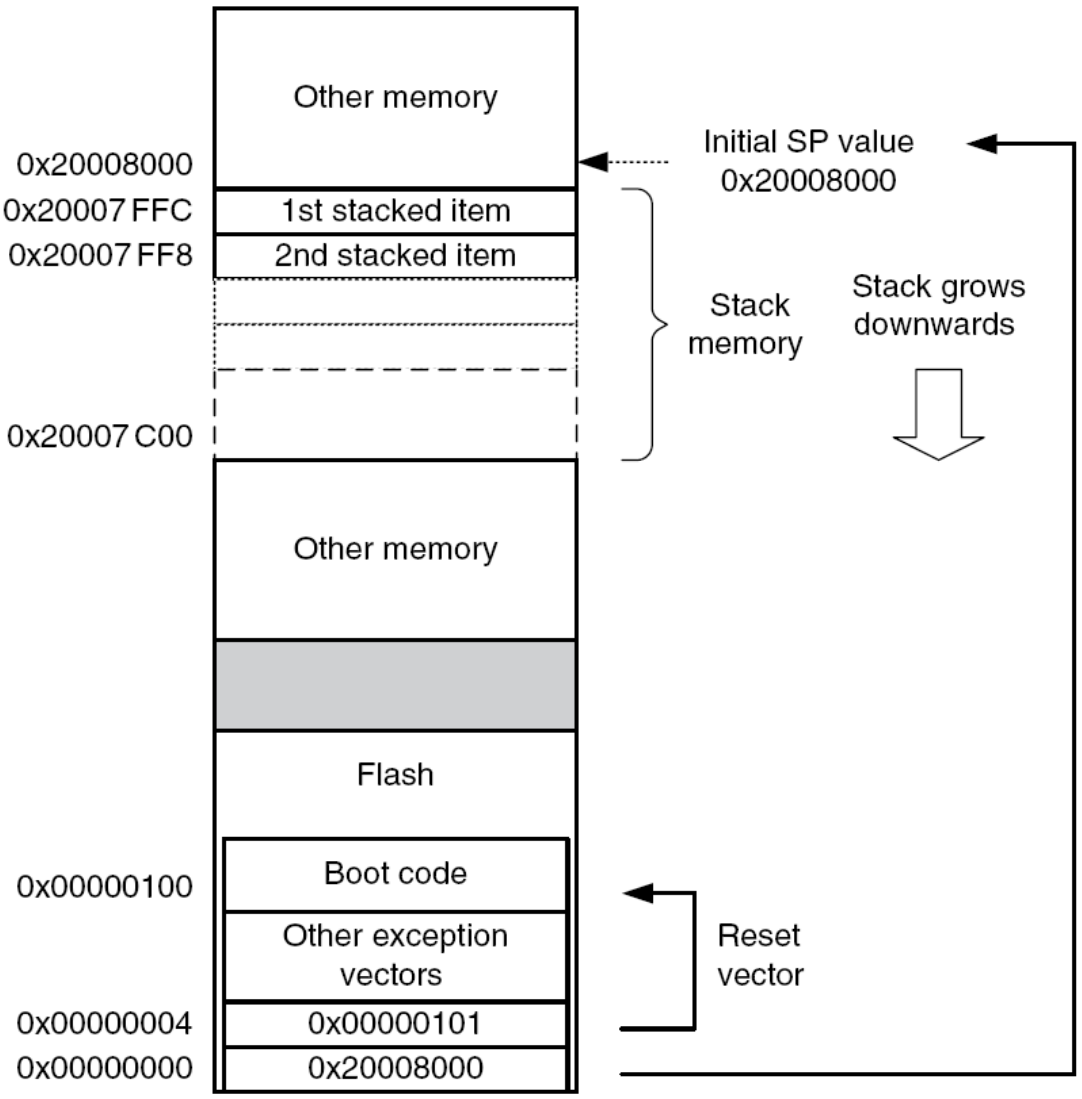
Table 7.7 Vector Table Offset Register (Address 0xE000ED08)

Bits	Name	Type	Reset Value	Description
29	TBLBASE	R/W	0	Table base in code (0) or RAM (1)
28:7	TBLOFF	R/W	0	Table offset value from code region or RAM region

ARM - CORTEX M3 (18)

E' interessante vedere come avvenga il reset. Il primo indirizzo è quello che contiene la posizione dello Stack, mentre il secondo quello della partenza del programma.





Qui abbiamo un esempio di come potrebbe essere localizzato lo Stack e l'inizio del programma.

ARM - CORTEX M3 (19)

Il numero di interruzione effettivamente disponibili dipende da come è stato implementato l'hd del micro, 256 è il massimo.

Ad ogni interruzione occorre dare una priorità, più o meno marcata. Per questo si devono impostare 8 bit di un registro, uno per ogni interruzione.

Il registro è a sua volta diviso in 2 parti: preempt-priority-field e subpriority-field:

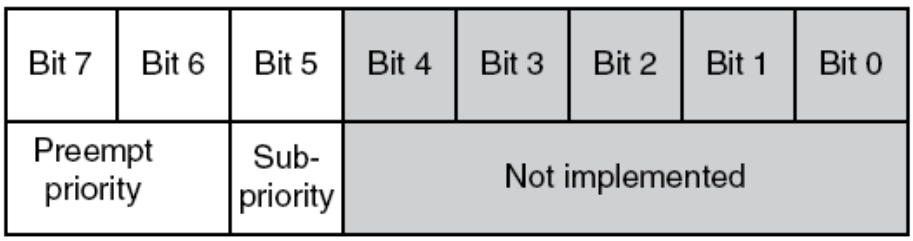
Table 7.5 Definition of Preempt Priority Field and Subpriority Field in a Priority Level Register in Different Priority Group Settings

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

Il preempt-priority-field è considerato per verificare se l'interruzione che si sta eseguendo deve essere sopraffatta da una nuova interruzione con priorità maggiore.

Il subpriority-field è considerato solo quando 2 interruzioni aventi la stessa preempt accadono contemporaneamente: viene eseguita quella con priorità maggiore.

Se le priorità preempt sono meno di 128 (il bit 0 non può essere usato nella preempt nel gruppo 0). Si toglie un numero di bit adeguato nella parte meno significativa, come nell'esempio qui, dove vengono usati solo 3 bit:



ARM - CORTEX M3 (20)

A questo punto occorre abilitare le interruzioni che servono. Le abilitazioni si attuano scrivendo nel registro Interrupt set Enable. Il registro è a 32 bit, per cui ve ne potranno essere più di uno: SETENA0, SETENA1, ecc.

Per disabilitare un'interruzione non bisogna scrivere nel registro SETENA, ma bensì nel registro Interrupt Clear Enable Register. Anche qui potremmo avere più di un tale registro: CLRENA0, CLRENA1, ecc.

Un altro registro utile è l' Interrupt Set-Pending Register: SETPEND0, SETPEND1, ecc. Questo registro ha una doppia funzionalità. Leggendolo siamo in grado di capire chi sta in coda. Scrivendoci possiamo forzare in attesa una interruzione manualmente. Se invece volessimo cancellare una interruzione in attesa dobbiamo scrivere nel registro CLRPEND0, CLRPEND1, ecc.

In C l'abilitazione dell'interruzione con l'impostazione della sua priorità si fa così:

```
NVIC_SetPriorityGrouping(5);
NVIC_SetPriority(7, 0xC0); // Set IRQ#7 priority level to 0xC0
NVIC_EnableIRQ(7);
```

IMPORTANTE: occorre includere nel progetto i file .h opportuni contenenti le funzioni menzionate.

La filosofia da adottare è capire dall'help che .h serve. Sviscerare il .h alla ricerca delle funzioni che servono. Tipicamente quella di inizializzazione, abilitazione, ecc.

Peculiare per queste istruzioni è anche il manuale CMSIS, che si può trovare a www.onarm.com.

ARM - CORTEX M3 (21)

E' possibile disabilitare tutte le interruzioni in un colpo mediante i registri PRIMASK e FAULTMASK. PRIMASK disabilita tutte le interruzioni dell'utente eccetto la NMI e hard fault. FAULTMASK blocca anche l'hard fault. Esempio di utilizzo in C:

```
void __enable_irq(); // Clear PRIMASK
void __disable_irq(); // Set PRIMASK
void __set_PRIMASK(uint32_t priMask); // Set PRIMASK to value
uint32_t __get_PRIMASK(void); // Read the PRIMASK value
```

Il registro BASEPRI disabilita tutte le priorità al di sotto del livello a cui viene impostato:

```
__set_BASEPRI(0x60); // Disable interrupts with priority
                     // 0x60-0xFF using CMSIS
```

Il BASEPRI può essere annullato con:

```
__set_BASEPRI(0x0); // Turn off BASEPRI masking
```

ARM - CORTEX M3 (22)

A questo punto occorre assegnare il sottoprogramma alle interruzioni. Questo viene realizato assegnando dei nomi prestabiliti che devono avere le funzioni. Per individuarli occorre andare nel file di configurazione del micro che deve essere allegato al progetto. Il file ha nome startup_nome_micro.s. Qui un esempio dei vettori:

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     SVC_Handler               ; SVCall Handler
                DCD     DebugMon_Handler          ; Debug Monitor Handler
                DCD     0                         ; Reserved
                DCD     PendSV_Handler            ; PendSV Handler
                DCD     SysTick_Handler           ; SysTick Handler

                ; External Interrupts
                DCD     WDT_IRQHandler            ; 16: Watchdog Timer
                DCD     TIMER0_IRQHandler         ; 17: Timer0
                DCD     TIMER1_IRQHandler         ; 18: Timer1
                DCD     TIMER2_IRQHandler         ; 19: Timer2
                DCD     TIMER3_IRQHandler         ; 20: Timer3
                DCD     UART0_IRQHandler          ; 21: UART0
                DCD     UART1_IRQHandler          ; 22: UART1
                DCD     UART2_IRQHandler          ; 23: UART2
                DCD     UART3_IRQHandler          ; 24: UART3
```

ARM - CORTEX M3 (23)

Quindi in C la gestione della UART1 può essere fatta così:

```
__irq void UART1_Handler(void) {
   ... // process IRQ request for the peripheral
   ... // Deassert IRQ request in peripheral
   return;
}
```

Naturalmente la UART è un caso particolare perché sia la ricezione che la trasmissione generano la stessa interruzione. Quindi va individuato l'autore della richiesta. Inoltre nella UART in genere è l'utilizzatore che dve annullare la richiesta e non è svolto automaticamente all'uscita dal sottoprogramma.

ARM - CORTEX M3 (24): suggerimenti

Come per l'8051 è importante l'inclusione del file che configura il micro nell'IDA. Per esempio, per il Cortex della serie LPC17xx di NXP, all'inizio del progetto si deve scrivere:

#include <LPC17XX.h>

Questo file appartiene poi alle librerie CMSIS di base del Cortex.

In questo file si trovano tutte le definizioni delle periferiche, definite ognuna come una struttura, struct, col nome:

LPC_nomeperiferica

I vari registri sono quindi indirizzabili con:

LPC_nomeperiferica->nome_registro

ad esempio, per impostare il match register 0 del timer 2, si cerca il nome del timer 2 nel file (ovvero LPC_TIM2) e poi si indirizza il registro con

LPC_TIM2->MR0 = 0x00asd;

Altra utilità: all'inizio del file c'è una tabella con i nomi di tutti le interruzioni.

I file come LPC17xx.h rispettano lo standard CMSIS. La loro documentazione si trova nella guida completa di uVision cercando "CMSIS Overview". Ancora, in CMSIS usage sono elencati i file da includere ed utilizzare.

Ad esempio, nel file core_cm3.h sono riportate tutte le istruzioni "di default" per gli interrupt. Si trovano anche alla voce "Nested Vectored Interrupt Controller" nella guida sempre di uVision.

ARM - CORTEX M3 (25)

Ecco un esempio di utilizzo di interruzione: la seriale. Cominciamo dalla sua inizializzazione:

```
/* Enable UART0 interrupt */
LPC_UART0->IER = 0x01 | 0x02 | 0x03;
NVIC_EnableIRQ(UART0_IRQn);
```

Ora invece la funzione chiamata quando si verifica l'vento:

```
void UART0_IRQHandler (void)
{
static char r;

switch ((LPC_UART0->IIR&0x0F)) { */IIR=registro evento/*
        case 0x01: /* no int pending, called by UART0_putstring? */  */ Si è chiamto l'UART in modo artificiale, abilitando il pending register/*
                if ((LPC_UART0->LSR>>5)&0x01){ /* check for empty THR: se vuoto ultima trasmissione terminata */
                        UART0_send_buffer(); /* start transmission     */
                }
                break;
        case 0x06:          /* error */
                // debug here?
                break;
        case 0x04:          /* data received */
                r = LPC_UART0->RBR;
                UART0_rec_buffer(r);
                break;
        case 0x0C:          /* data in time out */
                r = LPC_UART0->RBR;
                UART0_rec_buffer(r);
                break;
        case 0x02:          /* THR-Empty interrupt */
                if (UART0_Buffer->out_n == 0) {
                        /* if buffer is empty, do nothing            */
                        break;
                }
                UART0_send_buffer();
                break;
        }
}
```

Bibliografia

N.Storey,
ELECTRONICS A SYSTEMS APPROACH
Addison Wesley, 1992.

Bibliografia 8051:

M.Predko,
PROGRAMMING AND CUSTOMIZING THE 8051 MICROCONTROLLER
Mc Graw-Hill, 1999.

Ken Arnold
EMBEDDED CONTROLLER HARDWARE DESIGN
LLH Technology Publishing, 2000. Cod. Biblio: 621.3916 arnk.emb 2001

Bibliografia ARM:

Steve Furber
ARM
System-on-chip architecture, second edition
Addison-Wesley, 2000, cod. Biblio 004 165 URS ARM 2000BS (2 copie)

Trevor Martin
The Insider's Guide To The Philips ARM7-Based Microcontrollers
Hitex (UK) Ltd.

A.N.Sloss, D.Symes, C.Wright
ARM System Developer's Guide
Elsevier, 2004, Cod. Biblio 005.1 SLOA.ARM/2004

D.Seal
ARM Architecture Reference Manual (2nd Edition)
Addison-Wesley Professional, 2001, Cod. Biblio 004.22SEAD.ARM /2001

Bibliografia ARM CORTEX M3:

J. Yiu
The definitive guide to the ARM CORTEX-M3, second Edition
Elsevier 2010. Biblio 621.3916 YIUJ.DEF/2010.